SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs

Mohammad Mejbah ul Alam * Tongping Liu * Guangming Zeng† Abdullah Muzahid

University of Texas at San Antonio †Lingshan Road
San Antonio, TX 78249 Shanghai, China, 200135
{Mohammad.Alam,Tongping.Liu,Abdullah.Muzahid}@utsa.edu, zenggming@gmail.com

Abstract

Despite the obvious importance, performance issues related to synchronization primitives are still lacking adequate attention. No literature extensively investigates categories, root causes, and fixing strategies of such performance issues. Existing work primarily focuses on one type of problems, while ignoring other important categories. Moreover, they leave the burden of identifying root causes to programmers. This paper first conducts an extensive study of categories, root causes, and fixing strategies of performance issues related to explicit synchronization primitives. Based on this study, we develop two tools to identify root causes of a range of performance issues. Compare with existing work, our proposal, `SyncPerf`, has three unique advantages. First, `SyncPerf`'s detection is very lightweight, with 2.3% performance overhead on average. Second, `SyncPerf` integrates information based on callsites, lock variables, and types of threads. Such integration helps identify more latent problems. Last but not least, when multiple root causes generate the same behavior, `SyncPerf` provides a second analysis tool that collects detailed accesses inside critical sections and helps identify possible root causes. `SyncPerf` discovers many unknown but significant synchronization performance issues. Fixing them provides a performance gain anywhere from 2.5% to 42%. Low overhead, better coverage, and informative reports make `SyncPerf` an effective tool to find synchronization performance bugs in the production environment.

* Alam and Liu contributed equally to this work.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

DOI: http://dx.doi.org/10.1145/3064176.3064186

1. Introduction

Designing efficient multithreaded programs while maintaining their correctness is not an easy task. Performance issues of multithreaded programs, despite being the primary cause of more than 22% synchronization fixes of server programs [23], get less attention than they deserve. There are many prior works [8, 19, 21, 23, 28, 41, 43] in this domain, but none of them systematically investigates performance issues related to different types of synchronization primitives. Most existing works cannot identify root causes, and provide helpful fixing strategies.

This paper studies various categories, root causes, and fixing strategies of performance issues related to different synchronization primitives such as locks, conditional variables, and barriers. Lock/wait-free techniques and other mechanisms (such as transactional memory [20]) are not covered in this paper. The study divides synchronization related performance issues into five categories: *improper primitives, improper granularity, over-synchronization, asymmetric contention, and load imbalance* (shown in Table 1). The first four categories are related to various locks, whereas the last one is related to other synchronizations such as conditional variables and barriers. The study shows that the same symptom can be caused by multiple root causes. For example, high contention of locks can occur due to too many data items under the same lock, too-large critical sections, over-synchronization, or asymmetric lock contention (more details in Section 2). Without knowing the root cause, it is difficult for programmers to fix these bugs effectively. The study also shows that different categories of problems may have different symptoms and thus, different solutions. Finally, the study presents some ideas for identifying and fixing these performance issues. The study not only helps users to identify and fix synchronization performance issues, but also enables future research in this domain.

Prior work [8, 19, 21, 23, 28, 41, 43] focuses excessively on locks that are both acquired frequently and highly contended. Our first observation is that *performance problems can also occur with locks that are not excessively acquired*

or highly contended. This is shown in Figure 1. Existing work focuses on quadrant 2 or Q2. Locks of Q2 can definitely cause performance issues but they are not the only culprits. `SyncPerf` finds potential problems with the other two quadrants: (i) locks that are not acquired many times may slow down a program if the critical sections are large and potentially introduce high contention and/or a long waiting time (Q1); (ii) locks that are acquired excessively may cause significant performance problems, even if they are barely contended (Q4). Intuitively, locks of Q3 (lowly contended and not acquired many times) will not cause performance problems. Our second observation is that *it is not always sufficient to identify root causes of a problem based on the behavior of a single synchronization*. For example, for asymmetric contention where different locks are protecting similar data with different contention rates, we have to analyze the behavior of all those locks that typically have the same initialization and acquisition sites. By checking all of those locks together, we can notice that some locks may have higher contention and acquisition than others.

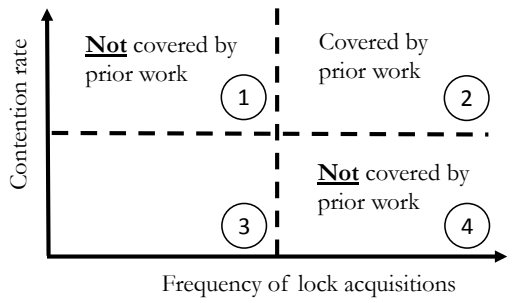

Figure 1. Existing work mainly focuses on the locks of quadrant 2 (Q2) while ignoring those in quadrant 1 (Q1) & 4 (Q4). Contention rate = what fraction of times a lock is found to be unavailable. Frequency of acquisitions = how many times a lock is acquired per second. Note that we use contention rate instead of lock waiting time as an axis since the later one is closely related to the first one.

Driven by these two intuitive but novel observations, we develop `SyncPerf` that not only reports the callsites of performance issues, but also helps diagnose root causes and suggests possible fixes for a range of performance issues related to synchronization primitives. Most existing works [8, 28, 41] just report the callsites of the performance issues (mostly high contention locks), while leaving the burden of analyzing root causes (and finding possible fixes) to programmers. The only work similar to ours was proposed by Yu et al. [43]. However, `SyncPerf` excels it by having a better detection ability (thanks to the novel observations), a broader scope, and much lower overhead (Section 6).

`SyncPerf` starts by monitoring the execution of an application and collecting information about explicit synchronization primitives. More specifically, it collects (i) for a lock, how many times it is acquired, how many times it is found to be contended, and how long a thread waits for the lock, (ii) for a try-lock, how many times it is called and how many times it fails because of contention, and finally (iii) for load imbalance, how long different threads execute, and how long they are waiting for synchronizations. `SyncPerf` also collects callsites for each synchronization operation and thread creation function to help pinpoint the actual problems. After this, `SyncPerf` integrates and checks the collected information to identify root causes: (i) it checks behavior of all locks with the same callsites to identify asymmetric contention issue, (ii) it computes and compares waiting time of different threads to identify load imbalance issue, and (iii) it checks individual as well as collective (based on callsites) information of locks (i.e., the number of acquisitions and number of times they are contended) to identify other performance issues. This integration is very important, and helps uncover more performance issues. `SyncPerf` is able to find more performance issues than any prior work (Table 2). For some of the problems, such as asymmetric contention, and load imbalance, `SyncPerf`'s detection tool automatically reports root causes. It also presents an optimal task assignment to solve load imbalance problems. For other problems, `SyncPerf` provides sufficient information as well as an informal guideline to diagnose them manually. `SyncPerf` also provides an additional optional tool (that programmers can use offline) to help the diagnosis process.

Contribution:

Overall, this paper makes the following contributions:

- This paper provides a taxonomy of categories, root causes, and fixing strategies of performance bugs related to explicit synchronization primitives. The taxonomy is useful not only to identify and fix synchronization performance problems but also to enable future research in this field.
- `SyncPerf` uses an intuitive observation that performance problems may occur even when locks are not frequently acquired or highly contended. There is no existing work that actually uses this observation. Due to this observation, `SyncPerf` finds many previously unknown performance issues in widely used applications.
- `SyncPerf` makes a novel observation that it is hard to detect problems such as asymmetric contention and load imbalance by observing the behavior of a single synchronization. To solve this problem, `SyncPerf` proposes to integrate information based on callsites of lock acquisitions (and initializations), lock variables, and types of threads. This integration also contributes to the detection of some unknown issues.
- Finally, `SyncPerf` provides two tools that help diagnose root causes of performance bugs. The first one is a detection tool that can report susceptible callsites and synchronization variables with potential performance issues, and

identify some root causes such as asymmetric contention and load imbalance. This tool has extremely low overhead (only 2.3%, on average). The tool achieves such low overhead even without using the sampling mechanism. The low overhead makes the tool a good candidate for the deployment environment. When multiple root causes may lead to the same behavior and thus, cannot be diagnosed easily, `SyncPerf` provides a heavyweight diagnosis tool that collects detailed accesses inside susceptible critical sections to ease the diagnosis process. Both of these tools are software-only tools that do not require any modification or recompilation of applications, and custom operating system or hardware support.

Outline:

The remainder of this paper is organized as follows. Section 2 presents a categorization of synchronization related performance bugs, and the workflow of `SyncPerf` tools; Section 3 describes the implementation details of `SyncPerf`; Section 4 presents the experimental results; Section 5 discusses some limitations of `SyncPerf`; Section 6 describes related work, and finally, Section 7 concludes the work.

2. Overview

In this Section, we first provide a categorization of performance issues related to synchronization primitives. We propose this categorization based on our experience and analysis of the bugs detected during experiments. Therefore, the categorization may not be exhaustive. However, the categorization serves as the basis for `SyncPerf`'s observations and design choices. Lastly, we show a workflow that describes the identification of root causes based on symptoms.

2.1 Categorization

Synchronization related performance issues can be divided into five categories (Table 1).

2.1.1 Improper Primitives

Programmers may use a variety of synchronization primitives (e.g., atomic instructions, spin locks, try-locks, read-/write locks, mutex locks etc.) to protect shared accesses. These primitives impose different runtime overhead, increasing from atomic instructions to mutex locks. The spin lock of `pthread` library, for example, incurs 50% less overhead than the mutex lock when there is no contention. However, during high contention, the spin lock may waste CPU cycles unnecessarily [1, 30].

Different synchronization primitives have different use cases. Atomic instructions are best suited to perform simple integer operations (e.g., read-modify-write, addition, subtraction, exchange etc.) on shared variables [9, 34]. Spin locks are effective for small critical sections that have very few instructions but cannot be finished using a single atomic instruction [1, 30]. Read/write locks are useful for read-mostly critical sections [26, 32]. Try-locks allow a program to pursue an alternative path when locks are not available [38]. Finally, mutex locks are used when the critical sections contain waiting operations (e.g., conditional wait) and have multiple shared accesses. Any deviation from the preferred use cases may result in performance issues.

Identification: Improper primitives (usually in Q2 and Q4) typically cause extensive try-lock failures or extensive lock acquisitions, but low to moderate contention. Extensive try-lock failures, where a try-lock fails immediately because the lock is held by another thread, indicate that we should use a blocking method that combines conditional variables with mutexes to avoid continuous trial. Extensive lock acquisitions may incur significant performance degradation even without high contention. The issue of improper primitives is ignored by existing work [8, 19, 23, 28, 41]. However, its importance can be seen from `facesim` application of PARSEC [3] where changing mutex locks to atomic instructions boosts performance by up to 30.7% (Table 2).

2.1.2 Improper Granularity

Significant performance degradation may occur when locks are not used with a proper granularity. There are several cases listed as follows.

1. If a lock protects too many data items (e.g., an entire hash table, as in the `memcached-II` bug of Table 2), the lock may introduce a lot of contention. Splitting a coarse-grained lock into multiple fine-grained locks helps improve performance.

2. If a lock protects a large critical section with many instructions, it may cause high contention and thus, a significant slowdown. `canneal` of PARSEC, for example, has a critical section that includes a random number generator. Only few instructions inside the critical section access the shared data. Although the number of acquisitions is only 15, performance is boosted by around 4% when we move the random generator outside the critical section.

3. If a critical section has very few instructions, then the overhead of lock acquisitions and releases may exceed the overhead of actual computations inside. In that case, the program can suffer from performance degradation [14]. One possible solution is to merge multiple locks into a single coarse-grained one.

Identification: Locks in the first two cases may incur significant contention. However, without knowing the memory accesses inside the critical section, it is hard to identify this type of problems manually. Therefore, `SyncPerf` provides an additional diagnosis tool that tracks all memory accesses protected by a specific lock. Programmers can use the tool offline after some potential problems have been identified by the detection tool. With the collected information, we can easily differentiate between the first two cases as described in Table 1. It is relatively hard to identify the third case.

Category	Symptoms	Quadrant	Root Cause	Solution
Improper Primitives	extensive lock acqs, low contention	Q4	small CS with a simple integer operation	atomic instructions
			small CS	spin locks
	extensive lock acqs, moderate contention	Q2 & Q4	read/write only CS	reader/writer locks
	extensive try-lock failures	N/A	N/A	mutex + cond. var.
Improper Granularity	moderate to high contention	Q1 & Q2	too many data items under the same lock	finer locks
			too large CS	shrinking CS
	extensive lock acqs, low contention	Q4	multiple small CS in the same function	coarser locks
Over-Synchronization	extensive lock acqs	Q2 & Q4	already mutually exclusive	removing unnecessary locks
			accessing local data only	removing unnecessary locks
	extensive lock acqs high contention	Q2	different locks are serialized by the same lock	removing the common lock
Asymmetric Contention	high contention locks	Q2	asymmetric contention rate	distribute contention
Load Imbalance	disproportionate thread computation and waiting time	N/A	N/A	task redistribution

Table 1. Categorization of synchronization performance issues. "CS" is short for "Critical Section" and "acqs" is for "acquisitions".

2.1.3 Over-synchronization

Over-synchronization indicates a situation where a synchronization becomes unnecessary because the computations do not require any protection or they are already protected by other synchronizations. This term is borrowed from existing work [23]. There are the following cases.

1. A lock is unnecessary if a critical section only accesses the local data, but not the shared data.

2. A lock is unnecessary if the protected computations are already atomic.

3. A lock is unnecessary if another lock already protects the computations. MySQL-5.1 is known to have such a problem [7, 23], which utilizes the `random()` routine to determine the spin waiting time inside a mutex. Unfortunately, this routine has an internal lock that unnecessarily serialize every thread invoking this `random()` routine. The problem has been fixed by using a different random number generator that does not have any internal lock for the fastmutex.

Identification: Over-synchronization problems can cause a significant slowdown when there are extensive lock acquisitions. This situation is similar to the first two categories of improper granularity issue. Therefore, our diagnosis tool (described in Section 3.2) may help analyze this situation. After a problem is identified, unnecessary locks can be removed to improve performance. However, removing locks may introduce correctness issue, and has to be done cautiously.

2.1.4 Asymmetric Contention

Asymmetric contention occurs when some locks have significantly more contention than others that protect similar data. This category is derived from "asymmetric lock" [10]. For instance, a hash table implementation may use bucket-wise locks. If the hash function fails to distribute the accesses uniformly, some buckets will be accessed more frequently than the others. Consequently, locks of those buckets will have more contention than the others. Coz [10] finds such a problem in `dedup`. Changing the hash function improves performance by around 12%.

Identification: To identify this type of problems, `SyncPerf` collects the number of lock acquisitions, how many times each lock is found to be unavailable, and their callsites. If multiple locks are protecting similar data (typically identified by the same callsites of lock acquisitions and releases), `SyncPerf` checks the lock contention rate and the number of acquisitions of these locks. When an asymmetric contention rate is found (e.g., when the highest contention rate is $2\times$ or more than the lowest one), `SyncPerf` reports an asymmetric contention problem. Asymmetric contention problem is reported automatically without any manual effort. Programmers, then, can fix the problem by evenly distributing the contention. Unlike `SyncPerf`, Coz relies on manual inspection to identify this type of problems.

2.1.5 Load Imbalance

A thread can wait due to synchronizations such as mutex locks, conditional variables, barriers, semaphores etc. A parent thread can also wait when it tries to join with the children threads. If a group of threads (i.e., threads with the same thread function) is found to have a waiting period much longer than that of other groups of threads, this may indicate a performance issue caused by load imbalance [12, 25, 33, 40].

Identification: To identify load imbalance problems, it collects the execution and waiting time of different threads by intercepting thread creations and synchronization functions. If the waiting time or computation time of different threads are substantially different (e.g., outside a certain range, say 20%), the program can be identified as having a load imbalance problem.

Finding an optimal task assignment: SyncPerf can suggest an optimal task assignment for load imbalance problems after the identification. It calculates the computation time of every thread by subtracting all waiting time (on conditional variables, mutex locks, and barriers) from their execution time. It then computes the total computation time of different groups of threads according to their thread functions, where threads executing the same function belong to the same group. In the end, SyncPerf suggests an optimal task distribution – each group of threads will be assigned an optimal number of threads that is proportional to the total workload of that type. Section 4.4.5 presents some examples showing how SyncPerf can suggest an optimal configuration for different types of threads to fix the load imbalance problems.

2.2 Workflow

The high level workflow of SyncPerf is shown as Figure 2. For mutex locks, SyncPerf reports locks inside 3 quadrants (Q1, Q2, and Q4 of Figure 1), while skipping Q3 locks that do not cause performance issues. Additionally, it reports try-lock failure rates and whether there is a load imbalance problem among different types of threads. For the load imbalance problem, SyncPerf not only reports the root cause but also suggests an optimal configuration for different types of threads. This is done without any manual intervention. Programmers can use the suggested distribution to fix the load imbalance problem. If there is an asymmetric contention problem among similar locks, the tool automatically identifies the root cause. However, it is up to the programmer to develop a possible fix.

After getting the behavior of locks in 3 quadrants, if the reported code segments are simple, programmers can easily inspect them manually to determine which category a problem belongs to and take corresponding actions. This can be as simple as consulting Table 1. For complex situation, our additional diagnosis tool can collect detailed information for critical sections that reported by our detection tool, in order to help programmers determine the particular type of performance issues. Again, Table 1 can be used as an informal guideline during the categorization process. After determining the type of performance bugs, Table 1 can guide programmers to develop a fix for the bug. Some of the fixing strategies (e.g., fixing of over-synchronization problem) might require programmers to carefully consider correctness issues.

3. Implementation Details

SyncPerf provides two tools to assist programmers in identifying bugs and fixing them: *a detection tool* and *a diagnosis tool*. By combining these two tools, SyncPerf not only answers "what" and "where" questions, but also "why" and "how to fix" (partially) questions for most synchronization related performance bugs.

The detection tool uses a lightweight profiling scheme to detect synchronizations with potential performance issues. It can also diagnose the root causes for asymmetric contention, extensive try-lock failures, and load imbalance problems without any manual intervention. The detection tool achieves a lower performance overhead than existing tools (even without using the sampling mechanism) [41]. Details of this tool are presented in Section 3.1.

The diagnosis tool is based on Pin [29], a binary instrumentation tool. The diagnosis tool monitors memory accesses inside specific critical sections to help identify root causes of problems with the same behavior. This heavyweight diagnosis tool is only employed when the detection tool reports some potential problems that cannot be diagnosed easily. It utilizes prior knowledge of the particular problems that are reported by the detection tool, and thus, instruments memory accesses inside the relevant critical sections only. Its overhead is about $6\times$ lower than the existing work that instruments all memory accesses [43].

3.1 Detection Tool

The challenge of SyncPerf is to collect data efficiently and analyze them effectively.

3.1.1 Collecting Data Efficiently

To collect the data, SyncPerf intercepts pthread's different types of explicit synchronization primitives, such as mutex locks, try-locks, conditional variables, barriers, and thread creation and exit functions, where the actual implementation is borrowed from the pthread library. This is similar to existing work [41]. However, SyncPerf outperforms them with a lower overhead and better detection ability.

SyncPerf intercepts pthread_create function calls and passes a custom function to the actual pthread_create function. This custom function calls the actual start_routine function, and collects timestamps of thread starting and exiting using RDTSC timer [22]. The timestamps are saved into a thread wrapper as shown in Figure 3(b).
SyncPerf utilizes the following mechanisms to achieve the *extremely low overhead*.

Indirection and per-thread data: To collect data for mutex locks, a possible approach (used by existing work [41]) is to store the actual profiling data for each mutex lock in a global hash table. Upon every mutex invocation, we can lookup the hash table to find the pointer to the actual data, and then update it correspondingly. However, this approach introduces significant overhead due to the hash table lookup (and possible lock protection) on every synchronization operation, and the possible cache coherence messages to update the shared data (true/false sharing effect) [16, 21]. This is especially problematic when there is a significant number of acquisitions.

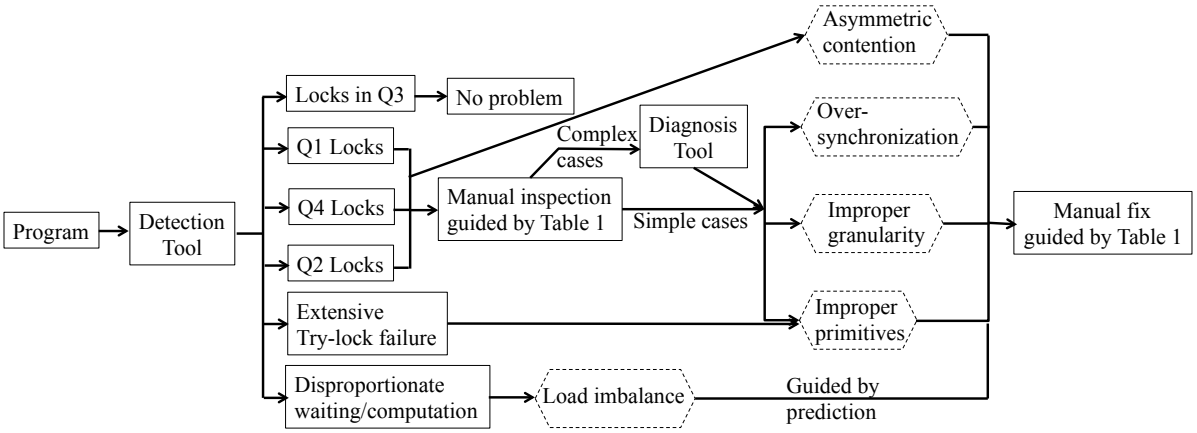

Figure 2. Workflow of `SyncPerf`.

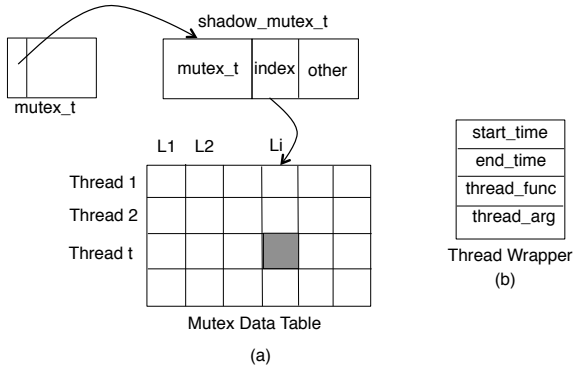

Figure 3. Data structures used by `SyncPerf`.

Instead, *`SyncPerf` uses a level of indirection to avoid the lookup overhead, and a per-thread data structure to avoid the cache coherence traffic. The data structure is shown in Figure 3(a)*. For every mutex, `SyncPerf` allocates a `shadow_mutex_t` object and uses the first word of the original `mutex_t` object as a pointer to this shadow object. The shadow mutex structure contains a real `mutex_t` object, an `index` for this mutex object, and some other data. The index is initialized during the initialization of the mutex, or during the first lock acquisition if the mutex is not explicitly initialized. This index is used to find an entry in the global *Mutex Data Table*, where each thread has a thread-wise entry. When a thread operates on a mutex lock, say L_i, `SyncPerf` obtains the `shadow_mutex_t` object by checking the first word of the original `mutex_t` object, and then finds its corresponding thread-wise entry using the `index` value. After that, the lock related data can be stored in its thread-wise entry, without generating any cache coherence message. Furthermore, `SyncPerf` prevents the false sharing effect by carefully keeping read-mostly data in `shadow_mutex_t` object and padding them properly [4, 27], while the actual profiling data (that keeps changing) is stored in thread-wise entries. The thread-wise data is collected and integrated in the reporting phase (Section 3.1.2).

Fast collection of callsites: `SyncPerf` collects callsite information of every synchronization operation to provide exact source code location of performance bugs. It is crucial to minimize the overhead of collecting callsites, especially when there is a large number of synchronization operations. `SyncPerf` makes three design choices to reduce the overhead. *First*, `SyncPerf` avoids the use of `backtrace` API of `glibc`, which is extremely slow due to its heavyweight instruction analysis. Instead of using `backtrace`, `SyncPerf` analyzes frame pointers to obtain call stacks efficiently. However, this can impose a limitation that `SyncPerf` cannot collect callsite information for programs without frame pointers. *Second,* `SyncPerf` collects call stacks up to the depth of 5. We limit the depth because deeper stacks introduce more overhead without any significant benefit. *Third,* `SyncPerf` avoids collecting already-existing callsites. Obtaining the callsite of a synchronization and comparing it against all existing callsites one by one (to determine whether this is a new one) may incur substantial overhead. Alternatively, `SyncPerf` utilizes the combination of the lock address and the offset between the stack pointer (`rsp` register) and the top of the current thread's stack to identify the call stack. When different threads invoke a synchronization operation at the same statement, the combination of the lock address and stack offset are likely to be the same. If a combination is the same as that of one of the existing callsites, `SyncPerf` does not collect callsite information. This method can significantly reduce the overhead of callsite collection and comparison.

Other mechanisms: To further reduce the runtime overhead, `SyncPerf` *avoids any overhead due to memory allocation* by preallocating the *Mutex Data Table* and a pool of shadow mutex objects. This is done during the program initialization phase. `SyncPerf` assumes a predefined but adjustable maximum number of threads and mutex objects for

this purpose. Also, *SyncPerf puts data collection code outside a critical section as much as possible to avoid expanding the critical section*. This avoids unnecessary serialization of threads.

Because of these careful design choices, SyncPerf imposes very low runtime overhead (2.3%, on average). Even for an application such as fluidanimate that acquires 40K locks per millisecond, SyncPerf imposes only 19% runtime overhead. Due to its low overhead, SyncPerf's detection tool can be used in production runs.

3.1.2 Analyzing and Reporting Problems

SyncPerf reports problems when a program is about to exit or it receives a special signal like SIGUSER2. SyncPerf performs two steps to generate a report.

First, it combines all thread-wise data of a particular synchronization together to check the number of lock acquisitions, lock contentions, and try-lock failures. It reports potential problems if any synchronization variable shows the behavior listed in Section 2.

Second, SyncPerf integrates information of different synchronization variables and threads together in order to discover more potential problems. (1) The behavior of locks with the same callsites are compared with each other: if some locks have significantly more contention than others, then there is a problem of asymmetric contention (Section 2.1.4). (2) Even if one particular lock is not acquired-many times, the total number of acquisitions of locks with the same callsite can be significant and thus, cause a severe performance issue. (3) SyncPerf integrates information of different threads together to identify load imbalance problems. When one type of threads (with the same thread function) have "disproportionate waiting time", it is considered to be a strong indicator for the load imbalance issue (Section 2.1.5). The integration of information helps find more potential problems.

3.2 Diagnosis Tool

The same behavior (e.g., lock contention) may be caused by different root causes, such as asymmetric contention, improper granularity, or over-synchronization. Therefore, SyncPerf provides a heavyweight diagnosis tool to help identify root causes of such problems. This heavyweight diagnosis tool is optional and not meant for production runs. Only when some potential problems are detected but they are hard to be diagnosed manually, this diagnosis tool may provide further information (e.g., memory accesses inside critical sections) that include: how many instructions are executed on average inside each critical section; how many of these instructions access shared and non-shared locations; how many different memory locations are accessed inside a critical section; and how many instructions are read or write accesses.

SyncPerf's diagnosis tool is based on a binary instrumentation framework, Pin [29]. It takes a list of problematic locks (along with their callsites) as the input, which is generated from the detection tool's report. When a lock function is encountered, it checks whether the lock is one of the problematic ones. If so, it keeps counting the instructions, and monitoring the memory accesses inside. The tool also maintains a hash table to keep track of memory locations inside critical sections. The hash table helps find out how many data items have been accessed inside a critical section. This information help identify the situation where a lock protects too many data items, or too many instructions that are accessing non-shared data inside a critical section. Like the detection tool, the diagnosis tool maintains thread-wise and lock-wise counters for each synchronization. It also integrates information together in the end.

4. Evaluation

This section will answer the following questions:

- **Usage Example:** What are the outputs of SyncPerf's tools? How we can utilize the report to identify root causes? (Section 4.2)
- **Bug Detection Ability:** Can SyncPerf detect real performance bugs related to synchronizations? (Section 4.3 and 4.4)
- **Performance Overhead:** What is the performance overhead of SyncPerf's detection and diagnosis tools? (Section 4.5)
- **Memory Overhead:** What is the memory overhead of the detection tool? (Section 4.6)

4.1 Experimental Setup

We performed experiments on a 16-core idle machine, with two-socket Intel(R) Xeon(R) CPU E5-2640 processors and 256GB of memory. It has 256KB L1, 2MB L2, and 20M L3 cache. The experiments were performed on the unchanged Ubuntu 14.10 operating system. We used GCC-4.9.1 with -O2, -g and -fno-omit-frame-pointer flags to compile all applications. SyncPerf utilizes the following parameters for the detection: contention rate larger than 10% is considered to be high, and the number of lock acquisition larger than 1000 per second is considered to be high. These thresholds are empirically determined. The parameters can be easily adjusted during the compilation of the detection tool. Section 4.3 evaluates false positives when using these parameters.

Evaluated Applications: We used a well-tuned benchmark suite, PARSEC [3], with native inputs. PARSEC applications have complexity comparable with real applications (see Table 3). We also evaluated three widely used real world applications: Apache, MySQL, and Memcached. We ran Apache-2.4 server program with the ab client that is distributed with the source code. We tested MySQL-5.6.27 using the sysbench client and the mysql-test. For mem-

```
Number of distinct locks : 1
-----------------------------------------
Locks with high contention, high frequency
-----------------------------------------
Total found: 0
-----------------------------------------
Locks with high contention, low frequency
-----------------------------------------
Total found: 1
Total waiting time(ms): 490
No.1 lock:
    Contention rate: 86.7%
    Acquisition frequency: 0.3
    Call site:
      ./rng.h:48

-----------------------------------------
Locks with low contention, high frequency
-----------------------------------------
Total found: 0
-----------------------------------------
Locks with asymmetric contention
-----------------------------------------
Total found: 0
```

Figure 4. Sample report of SyncPerf's detection tool.

```
Total information on 15 CSs of lock #1:
  # of instructions: 47109
  # of instructions accessing memory: 46979
  # of instructions accessing shared memory: 28
```

Figure 5. Sample report of SyncPerf's diagnosis tool.

```
  pthread_mutex_lock(&seed_lock);
  _rng = new MTRand(seed++);
  pthread_mutex_unlock(&seed_lock);
```

Figure 6. Corresponding code for canneal.

cached, we evaluated two different versions – memcached--1.4.4, and memcached-2.4.24, which are all exercised using the memslap benchmark. Data presented in the paper is for memcached-2.4.24 unless otherwise mentioned.

4.2 Usage Examples

SyncPerf provides two tools that help identify the root causes of problems. This section shows a usage example for application canneal of PARSEC.

Figure 4 shows an example report generated by SyncPerf's detection tool. For locks, it reports the results of three quadrants as shown in Figure 1. For each lock, SyncPerf reports source code information. For canneal, SyncPerf only reports one lock with high contention rate and low acquisition frequency in rng.h file. The corresponding code is shown in Figure 6. It is not very easy to understand this case. Therefore, we can resort to SyncPerf's diagnosis tool.

The diagnosis tool takes the reported locks from a specified file in the same directory, mostly call stacks of corresponding locks, as the input. An example of the report is shown in Figure 5. For canneal application, SyncPerf's diagnosis tool reports that only less than 1% instructions access the shared memory. Further consultation of the source code indicates that seed is the only shared access inside the critical sections. However, canneal currently puts the whole random generator inside the critical section, as described in Section 4.4.3. Moving the random generator out of the critical section improves the performance of this application by 4%.

4.3 Effectiveness

SyncPerf is effective in detecting synchronization related performance bugs. The results are shown in Table 2. SyncPerf detected nine performance bugs in PARSEC and six performance bugs in real world applications. Among the 15 performance bugs, **seven** were previously undiscovered, including **three** in large real applications such as MySQL and memcached. We have notified programmers of all of these new performance bugs. The MySQL-I bug does not exist any more because the corresponding functions have been removed in a later version (MySQL-5.7). Remaining bugs are still under review.

False Positives: We evaluated false positives of SyncPerf, using the threshold for contention rate and acquisition frequency of 10% and 1000 per second respectively. SyncPerf has no false positives for 12 programs (Table 2) of PARSEC and Memcached application. SyncPerf reports two potential performance problems in Apache. We have fixed one of them, with around 8% performance improvement. SyncPerf reports another one with high acquisition frequency (1252 per second) and low contention rate (4.5%). This is related to one_big_mutex of the queue. Fixing this problem requires significant changes in code. Therefore, we did not solve this problem. For MySQL, SyncPerf reports three potential performance bugs. Two of them have been fixed with performance improvement of 19% and 11% respectively. Another bug is related to $keycache' scache\_lock$. This lock has high acquisition frequency (1916 per second) and low contention rate (0.0%). We tried to use spin_lock as a replacement for the mutex lock, but we did not achieve any performance improvement. Therefore, this could be a potential false positive of SyncPerf. Thus, SyncPerf reports only two potential false positives at most.

False Negatives: It is difficult to assess whether SyncPerf has any false negative or not since there is no oracle that provides a complete list of all performance bugs in the evaluated applications. One option would be to experiment with known performance bugs. Our results indicate that SyncPerf detects all known performance bugs from the evaluated applications.

Category	Bug Id	Acq. Frequency	Contention Rate	Speed Up (%)	New Bug
Improper Primitives	facesim	15288	4.6	30.7	✓
	fluidnanimate-I	44185K	0	11.9	✓
	fluidanimate-II	16204	76.6	2.5	
	streamcluster	1199	69	3	
	x264	15857	0	8.5	✓
	Apache	49607	0	7.8	✓
	memcahced-I	117000	0	3.7	✓
Improper Granularity	canneal	0.3	86.7	4.0	✓
	memcached-II	71405	45.8	16.3	
	MySQL-I	723K	25.5	18.9	✓
	MySQL-II	146299	38.5	10.9	
Over-Synchronization	memcached-III	65445	0	3.0	
Asymmetric Contention	dedup-I	23531	13.6	12.1	
Load Imbalance	dedup-II	N/A	N/A	28	
	ferret	N/A	N/A	42	

Table 2. Effectiveness Results of `SyncPerf`'s detection. `SyncPerf` detects seven unknown bugs (with a mark in last column) in addition to nine known bugs. For `MySQL` and `Memcached`, the throughput is used as the performance metric. If an application has multiple bugs, we append a number as the bug id. "Acq. Frequency" column shows the number of acquisitions per second. The performance results are based on the average runtime of 10 executions.

4.4 Case Studies

This section provides more details about the detected performance bugs.

4.4.1 Extensive Acquisitions and High Contention

Existing tools [8, 19, 23, 28, 41, 43] mainly focus on performance bugs with this external symptom. However, only 4 out of 15 detected bugs have this symptom and they belong to three different categories as described below.

Asymmetric Contention: `dedup` is a compression program with data de-duplication algorithm. It has extensive lock acquisitions (23531 per second) and a high contention rate (13.6%) in an array of locks (`encoder.c:1051`). These locks protect different buckets of a hash table. `SyncPerf` detects these locks with asymmetric contention problems: these locks (with the same callsite) have different number of lock acquisitions, ranging from 3 to 8586; the one with the most acquisitions has a contention rate of 13.6%, while others have less than 1% contention rate. This bug is detected by Coz, but that requires expertise to identify the root cause [10]. Instead, `SyncPerf` can automatically identify this bug, without resorting to manual expertise. By changing the hash function to reduce hash collision using the prosed fix by the Coz paper, the performance is improved by 12.1%.

Improper Granularity: `Memcached-1.4.4` has a known performance bug caused by improper granularity of locks. It uses a single `cache_lock` to protect an entire hash table [13]. When we used `memslap` to generate 10000 `get` and `set` requests to exercise `Memcached` (with 16 threads), `SyncPerf` detects 71405 lock acquisitions per second and a high contention rate (45.8%). The diagnosis tool finds that a single lock protects over 9 million different shared locations. Clearly, this lock is too coarse-grained. Changing the global `cache_lock` to an array of `item_lock` as appeared in `Memcached-2.4.24` improves the throughput by 16.3%. This bug is shown as `memcached-II` in Table 2.

`MySQL`, a popular database server program, has a similar problem (MySQL-II in Table 2) [2]. When the input table data is not using the default character set of the server or `latin1`, `MySQL` calls `get_internal_charset()` function. `SyncPerf` detects extensive lock acquisitions (146299 per second) and a high contention rate (38.5%). Furthermore, `SyncPerf`'s diagnosis tool reports that a single mutex lock protects 512 different shared variables, with 16384 bytes in total. By replacing the lock with an array of locks with one lock per `charset` [2], the throughput of `MySQL` is improved by 10.9%.

`SyncPerf` reports a **new** performance bug (MySQL-I) in `end_thr_alarm` function of `MySQL`. `SyncPerf` reports extensive lock acquisitions (723K per second) and a high contention rate (25.5%) for mutex `LOCK_alarm`. The critical section has unnecessary conditional waits inside, possibly caused by code evolution. Programmers might have restructured the code logic, but forgot to remove these unnecessary waits. Removing the conditional wait improves performance of `MySQL` by 18.9%. We have reported this problem to programmers of MySQL and they replied that the corresponding code has been removed in MySQL-5.7.

4.4.2 Extensive Acquisitions but Low Contention

These locks are in Q4 of Figure 1 and are practically ignored by existing tools. As shown in Table 2, 5 out of 15 performance bugs fall into this category. They are **new** performance bugs detected by `SyncPerf`.

Improper Primitives: `facesim` is a PARSEC application that simulates the motion of human faces. `SyncPerf` detects that one type of locks (ones with the same callsite) has 15288 acquisitions per second but the contention rate is very low

```
- mutex = new pthread_mutex_t *[numCells];
+ int cache_item_count = CACHE_SIZE
    /sizeof(pthread_spinlock_t);
+ locks = new pthread_spinlock_t *[numCells];
  ......
  ......
  if(border[index]){
        pthread_mutex_lock(&mutex[index][j]);
+       pthread_spin_lock(&locks[index][j]);
        cell.density[j] += tc;
-       pthread_mutex_unlock(&mutex[index][j]);
+       pthread_spin_unlock(&locks[index][j]);
```

Figure 8. Fix for `fluidanimate-I`.

(4.6%). We replaced mutex locks and conditional variables with atomic instructions, and that improved the performance by 31%. A code snippet of fix is shown in Figure 7.

`fluidanimate` simulates fluid dynamics for the animation purpose in PARSEC. This application uses a two-dimensional array of mutex locks to synchronize concurrent updates of grid data. There are 92K distinct locks, with around 40M acquisitions per second. However, contention rate is almost 0%. In this application, each individual lock has only few thousand acquisitions, but one callsite has a combined acquisitions of 400M. `SyncPerf` detects this bug (`fluidanimate-I` in Table 2) by integrating the data from the same callsites (our second observation). Existing tools lack this ability and cannot find this problem. Manual inspection of the code shows that each critical section has less than 2 instructions. Therefore, we replaced these locks with `pthread_spin_lock` (Figure 8) and (in some cases) atomic instructions. The fix improved the performance by 11.9%.

`x264` is an application of PARSEC for video encoding. `SyncPerf` detects extensive lock acquisitions (15857 times per second), but with almost 0% contention rate. The diagnosis tool further shows that one critical section has less than 3 instructions. By replacing the existing code with atomic instructions (less than 5 lines of code change), 8.5% performance improvement is achieved.

`SyncPerf` reports a **new** performance bug in `Apache`. It detects that `g_timer_skiplist_mtx` mutex in `event.c-:1592` has a high acquisition frequency (49607 per second) with almost 0% contention rate. Replacing `pthread_mutex_lock` with `pthread_spinlock` results a 7.8% performance improvement.

The `memcached-I` bug in Table 2 also has extensive lock acquisitions but almost no contention. `SyncPerf`'s diagnosis tool identifies that there are only (on average) 2.7 instructions for every critical section. Thus, it is easy to fix this bug by replacing `pthread_mutex_lock` with `pthread_spin_lock`. By doing so, the performance is improved by 3.7%.

Over-Synchronization: The `memcached-III` bug in Table 2 has 65445 lock acquisitions per second and contention rate is almost 0%. This is a known over-synchronization bug [10]. In this application, `item_remove` function uses

```
        pthread_mutex_lock(&seed_lock);
-       _rng = new MTRand(seed++);
+       int x = seed++;
        pthread_mutex_unlock(&seed_lock);
+       _rng = new MTRand(x);
```

Figure 9. Fix for `canneal`.

`item_lock` to synchronize the removal operations. However, removal operations eventually execute an atomic instruction to decrement a reference count. By eliminating this unnecessary lock as suggested by prior work [10], we improved performance of this program by 3.0%.

4.4.3 Few Lock Acquisitions but High Contention

Usually, it is assumed that few lock acquisitions will not cause any performance problem. But this is not always true especially when contention causes threads to wait for a long time.

Improper Granularity: `SyncPerf` finds such a problem in *canneal*, which simulates a cache-aware simulated annealing algorithm to optimize the routing cost of a chip design. `canneal` acquires `seed_lock` only 15 times, one for each thread, but lock contention rate is 86%. Also, the total waiting time for this lock is around 0.5 seconds. As shown in Section 4.2, the root cause of this bug is not very obvious. The diagnosis tool further discovers that there are 46979 instructions accessing memory inside the critical sections, but only 28 instructions access the shared variable. By moving random number generator out of the critical section, the fix as shown in Figure 9 reduces the total execution time by around 2 seconds (from 51 seconds), and improves the performance of `canneal` around 4%. We are not very clear why reorganizing a critical section with 0.5 second waiting time can reduce the total execution time by around 2 seconds.

4.4.4 Extensive Try-lock Failures

As described in Section 2, too many try-lock failures indicate that a synchronization method combining mutex locks with conditional variables can be useful to improve the performance.

Improper Primitives: Both `fluidanimate` and `streamcluster` have this type of problems. For `fluidanimate-II` bug of Table 2, `SyncPerf` detects a high try-lock failure rate (76.6%) and 16204 lock acquisitions per second, locating at line 153 of `pthreads.cpp`. PARSEC implements a custom barrier by doing a busy wait with a try-lock. By replacing try-lock based custom barrier implementation with the `pthread`'s barrier, the performance of this program improves around 2.5%. `streamcluster` uses the same custom barrier as that of `fluidanimate`. We did not observe any performance improvement by replacing the custom barrier with `pthread`'s barrier. However, by modifying the custom barrier implementation with atomic instructions, we improved performance by 3%.

```
-static void waitForTasks( void) {
+static void waitForTasks(unsigned long seqnum) {
     TRACE;
 #ifdef ENABLE_PTHREADS
-  pthread_mutex_lock(&(sync.lock));;
-  sync.threadCount++;
-  if ( sync.threadCount == numThreads)
-  thread_cond_broadcast(&sync.tasksDone);;
-    pthread_cond_wait(&sync.taskAvail,&sync.lock);  pthread_mutex_unlock(&sync.lock);;
+   if(__atomic_add_fetch(&sync.threadCount, 1, __ATOMIC_RELAXED)  == numThreads) {
+      __atomic_store_n(&sync.areTasksDone, 1, __ATOMIC_RELAXED);
+    }
+    // Progress only when the sequence number is sufficient
+    while(__atomic_load_n(&sync.seqnum, __ATOMIC_RELAXED) != seqnum) { ; }
```

Figure 7. Fix for `facesim`.

4.4.5 Disproportionate Waiting/Computation

SyncPerf detects known load imbalance problems in two applications of PARSEC – dedup and ferret [10, 33]. For the load imbalance problems, SyncPerf also suggests an optimal task assignment to fix the problem, which is not possible in existing tools.

ferret searches images for similarity. ferret has four different stages that perform image segmentation, feature extraction, indexing, and ranking separately. By default, ferret creates the same number of threads for different stages. SyncPerf detects that different types of threads have a completely different waiting time, such as $4754ms$, $5666ms$, $4831ms$, and $34ms$ respectively. This clearly indicates that some stages may not have enough threads and others may have too many threads. SyncPerf predicts the best assignment to be (1-0.2-2.5-12.2). Thus either (1-1-3-11) or (1-1-2-12) can be an optimal distribution. We experimented with all possible assignments and found the best assignment to be (1-1-3-11). Using the suggested task assignment (1-1-3-11) improves the performance of ferret by 42%.

dedup creates the same number of threads for fine-grained fragmentation, hash computation, and compression stage. SyncPerf detects that the average waiting time of different groups of threads are $1175ms$, $0ms$, and $1750ms$ respectively (shown as dedup-II in Table 3). The average execution time of these groups of threads are $3723ms$, $1884ms$, and $12836ms$. Thus, the big difference between different groups of threads clearly indicates a load imbalance problem. SyncPerf predicts the best assignment to be (2.4-1.2-8.4). Thus, the assignment (2-1-9) or (3-1-8) or (2-2-8) could be the best assignment. The actual best assignment is (1-1-10), with 28% performance improvement. SyncPerf predicts the one just close to the best one, with 25% performance improvement.

4.4.6 Scalability of Fixes

Changing mutex locks to atomic sections may compromise the scalability of applications. To confirm whether the fixes are scalable or not, we ran our experiments with 16 threads and 32 threads on the machine with 16 cores. Figure 10 shows results for every bug fix, except two programs with load imbalance problems. Overall, our fixing strategies boost the performance for every bug, even with twice as many threads. However, it is worth noting that these fixes, especially the one that replaces mutex locks with spin locks, may experience some scalability problem when the number of threads are much larger than the number of cores, such as more than 4 times. This is due to the fact that spinning may waste CPU cycles unnecessarily.

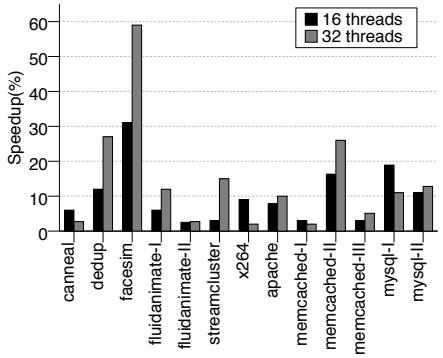

Figure 10. Scalability of fixes.

4.5 Performance Overhead

4.5.1 Detection Tool

The performance overhead of SyncPerf's detection tool is shown in Figure 11, with 16 threads in total. We ran every program 10 times and showed the average runtime in this figure. The execution time of these applications are normalized to that of using the pthread library. Higher bars indicate larger performance overhead. The deviation bars are not recognizable in the figure since the deviation of results is less than 0.1%. On average, SyncPerf introduces only 2.3% performance overhead. To the best of our knowledge, SyncPerf has the lowest overhead among similar tools, which is even faster than a tool with some sampling mechanism [41]. Except for two applications, fluidanimate

and dedup, SyncPerf's detection tool introduces less than 6% performance overhead. SyncPerf introduces a slightly higher performance overhead when the number of lock acquisitions per second is large and/or memory consumption is high. As shown in Table 3, fluidanimate is an extreme case with 92K distinct locks and more than 1,700M lock acquisitions in 40 seconds. SyncPerf only adds 19% overhead even for this application. Dedup introduces around 6.7% overhead because it has 62K lock acquisitions per second.

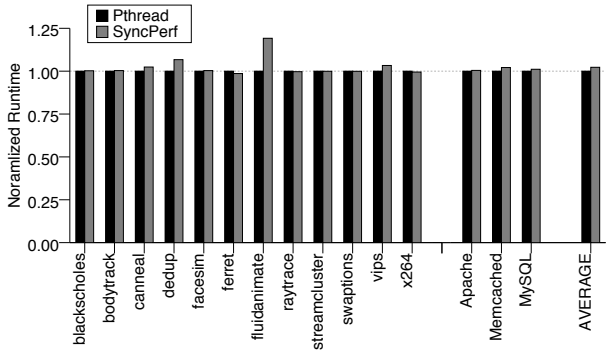

Figure 11. The normalized performance of SyncPerf.

4.5.2 Diagnosis Tool

Not all applications require the diagnosis tool. Sometimes it is fairly easy for programmers to recognize memory accesses inside critical sections. Programmers may use the diagnosis tool to obtain detailed memory accesses inside critical sections, when the critical sections are hard to be analyzed manually. In our experiments, we ran the diagnosis tool only for four applications - fluidanimate, canneal, Memcached, and MySQL. Among them, the highest overhead is 11.7 $\times$ for fluidanimate. Other applications such as MySQL, memcached, and canneal introduce 9.9$\times$, 8.7$\times$ and 3.5$\times$ overhead respectively. However, the overhead is significantly less than the existing work that uses Pin [43] (which can be up to 100$\times$ slower). SyncPerf avoids the excessive performance overhead by only checking accesses of the specified critical sections.

4.6 Memory Overhead

The physical memory overhead of SyncPerf's detection tool is listed in the last column of Table 3. We use the maximum physical memory consumption for the comparison, which is obtained through /proc/self/smaps file. We periodically collected this file and computed the the sum of proportional set size (PSS).

Table 3 shows that the memory overhead of SyncPerf varies from 1% to 215%. SyncPerf imposes some startup overhead for all applications, thus applications with small memory footprint tends to have a larger percentage of memory overhead, such as swaptions. We also notice that an application with more distinct locks has more memory overhead. However, SyncPerf only requires 36% more memory than that of pthread for all applications, which is acceptable as a tool.

5. Limitations and Future Work

SyncPerf has some limitations:

First, SyncPerf cannot identify performance bugs due to ad hoc synchronizations [42], atomic instructions or transactional memory [20]. Currently, it only focuses on performance problems related to explicit synchronization primitives. More specifically, the current implementation only supports POSIX APIs related to synchronizations, and is only verified on the Linux. However, the same idea can be easily applied to other threading libraries.

Second, SyncPerf cannot check contention of internal locks inside the glibc library. This can be fixed if the implementation is embedded inside the glibc library.

Finally, when there is no frame pointer inside a program's binary, SyncPerf may need to use backtrace to acquire callsite information or the program may requires the recompilation. The first method may incur more overhead for its detection tool.

In future, we would like to extend our tools to overcome some of these limitations. In addition, we would like to include a graphical interface so that some visual representation of results can be provided.

6. Related Work

Lock Contention Detectors: Thread Profiler [5] shows computation time, overhead, and blocking time along the critical path. Thus, programmers can focus on highly contented locks. IBM lock analyzer [24], HPROF [35], and JProfiler [17] report the acquisition and contention information for monitors/locks, and time spent on distinct locks. Lockmeter [6] records the overall statistics of each spin lock in Linux kernel, while ignoring other locks. Solaris performance analyzer shows time spent in locks (both asleep and spinning in user mode) and counts lock acquisitions and different kinds of sleep/wake transitions [36]. Although we can use such profilers to collect synchronization related information, we will not be able to detect all the problems without the insights of this paper. Moreover, SyncPerf detection tool can detect some of the problems automatically, without requiring manual expertise. Last but not least, SyncPerf contributes with an additional tool to help diagnose some specific type of performance issues. Zheng et al. [44] replay traces to identify lock contentions and calculate potential improvement that can be achieved by reducing the contentions. HaLock [21] utilizes a hardware memory tracing tool to store profiling data, in order to avoid memory interference with the applications. SyncPerf achieves better performance without any special hardware support, and detects more problems, not just lock contention. Tallen et al. [41] attributes lock contention to the holders of locks. However,

Application	LOC (#)	Runtime (Second)	Distinct Locks (#)	Total Acqs (#)	Contentions (#)	Cond waits (#)	Original Memory (MB)	SyncPerf Memory (MB)
blackscholes	374	43	0	0	0	0	610	619 (1.4%)
bodytrack	7773	26	7	1858971	7435	34236	30	74 (147%)
canneal	2812	51	1	15	12	0	941	981 (4%)
dedup	23565	16	2199	999467	60217	7	1616	1873 (16%)
facesim	34187	68	17	5035705	49604	1070249	326	364 (12%)
ferret	25758	5	5	2594	4	1146	68	111 (63%)
fluidanimate	2800	40	92396	1723580805	255053	35251	408	1064 (161%)
raytrace	13751	107	16	9431	480	3215	1077	1127 (4%)
streamcluster	3239	58	7	8088948	5557955	21251	108	278 (158%)
swaptions	1099	30	0	0	0	0	7	16 (115%)
vips	105260	18	3	2921	483	1	44	45 (2%)
x264	33024	29	35	202776	7	8236	481	1518 (215%)
Apache	253340	6	9707	56274	36	9685	50	71 (42%)
Memcached	10925	8	21	490000	78719	1	67	150 (125%)
MySQL	1606855	5	157	101280	3164	0	961	971 (1%)

Table 3. Characteristics of the evaluated applications.

SyncPerf achieves similar or even lower overhead without any sampling. Most importantly, unlike SyncPerf, their tool does not consider Q1 & Q4 locks and load imbalance among threads. Liu et al. [28] extends the blame shifting technique to OpenMP programs and reduces its memory overhead. Lock Visualizer [37] shows all call stacks responsible for synchronization issue in the current visible time range, sorted by cumulative time spent inside. This technique pinpoints synchronization with the highest contention. David et al. [18] defines Critical Section Pressure as the ratio of lock acquiring time and thread running time and finds performance problems related to locks. SyncProf [43] utilizes Pin to detect, localize, and optimize the synchronization performance problems. Like other works, it focuses only on highly contended locks and ignores Q1 & Q4 locks. It uses a graph representation to determine performance impact of individual critical sections. It proposes only three possible fixes whereas as shown in Table 1, SyncPerf proposes a taxonomy of a number of fixes. Finally, due to a large runtime overhead (up to 100×), SyncProf cannot be used in production environments.

Critical Thread Detectors: Chen et al. [8] quantitatively evaluates the impact of critical sections on the critical path of an application. Thus, they are able to identify locks most likely to have an impact on performance. DuBois et al. [15] propose to detect critical threads by proposing a new metric. The metric is calculated using some custom hardware support. They propose to improve performance of the critical thread using frequency scaling.

Load Imbalance Detectors: Tallent et al. [40] propose a postmortem load imbalance analysis technique to interpret call path profiles collected during runtime. The analysis may locate code regions responsible for communication wait time in SPMD programs. Resch et al. [12] compute a measure of severity of load imbalance for large-scale parallel applications. DeRose et al. [11] propose relative and absolute imbalance metrics to improve the detection ability of the previous one. Carnival measures performance for SPMD messaging programs and infers cause-and-effect for waiting time [31]. However, these techniques cannot predict the optimal solution as SyncPerf does. Navarro et al. [33] proposes a queueing theory based analytical model to characterize the performance of pipeline parallel applications like ferret and dedup. They propose parallel stage collapsing and dynamic workstealing scheduling to solve the load imbalance problem. Suleman et al. [39] uses hill climbing algorithm to select core assignment dynamically for pipeline stages in a pipeline parallel application. SyncPerf has better efficiency than the last one. Moreover, it is also very accurate in its prediction.

General Profiling Tools: Sometimes, general profiling tools can also detect performance problems related to synchronizations. Coz [10], a state-of-the-art profiling tool for multithreaded programs, may even estimate the performance impact of a statement or a code region. However, Coz will miss problems that involve multiple statements, especially when these statements are scattered over different code regions. SyncPerf, on the other hand, will integrate information of a lock with different callsites, or different locks with the same callsite, to provide better diagnosis of problems. SyncPerf's diagnosis tool even provide helpful suggestions for fixes. Such suggestions are not provided by general profiling tools. Therefore, SyncPerf can complement any general profiling tool.

7. Conclusion

This paper presents a taxonomy of categories, root causes, and fixing strategies of different performance issues related to synchronization primitives. We make two intuitive but novel observations and develop two tools (based on the observations) to uncover issues and root causes of different types of synchronization performance bugs. The first tool is an extremely low overhead (2.3%, on average) detection tool that detects potential performance issues for programs employing explicit synchronizations. Unlike existing tools, SyncPerf integrates information based on callsites of locks,

lock variables, and types of threads. Such integration helps uncover many synchronization performance issues. The second tool collects detailed memory accesses inside critical sections to help programmers determine root causes of complex problems. Overall, `SyncPerf` finds many previously unknown but significant synchronization performance bugs in widely-used applications. Low overhead, better coverage, and effectiveness make `SyncPerf` an appealing tool for the production environment.

Acknowledgements

We would like to thank our shepherd, Tim Harris. We are also thankful to Shan Lu, Charlie Curtsinger, Guoliang Jin, Harry Xu, Corey Crosser, Jinpeng Zhou, Hongyu Liu, Sam Silvestro, and anonymous reviewers for their valuable suggestions and feedbacks that helped improve this paper. The work is supported by UTSA, Google Faculty Award, and the National Science Foundation under Grants No. 1566154 and 1319983. The opinions, findings, conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

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

[2] Alexander Barkov. "thr_lock_charset global mutex abused by innodb". `https://bugs.mysql.com/bug.php?id=42649`, 2009.

[3] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[4] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[5] Clay P. Breshears. Using intel thread profiler for win32* threads: Philosophy and theory. `https://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory`, February 2011.

[6] Ray Bryant and John Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the linux®kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 17–17, Berkeley, CA, USA, 2000. USENIX Association.

[7] Mark Callaghan. "fast mutexes in mysql 5.1 have mutex contention when calling random()". `https://bugs.mysql.com/bug.php?id=38941`, 2008.

[8] Guancheng Chen and Per Stenstrom. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 71:1–71:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[9] GCC community. "built-in functions for memory model aware atomic operations". `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`, 2015.

[10] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. ACM.

[11] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 150–159. Springer Berlin Heidelberg, 2007.

[12] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray performance analysis tools. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[13] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[14] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.*, 49(2):218–244, March 1998.

[15] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 511–522, New York, NY, USA, 2013. ACM.

[16] S.J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[17] ej-technologies GmbH. Jprofiler: The award-winning all-in-one java profiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[18] David Florian. "continuous and efficient lock profiling for java on multicore architectures". `http://www-public.tem-tsp.eu/~thomas_g/research/etudiants/theses/david-phd-thesis.pdf`, 2015.

[19] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 426–438, New York, NY, USA, 2015. ACM.

[20] Maurice Herlihy and J. Eliot B. Moss. Transactional mem-

ory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[21] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. Halock: Hardware-assisted lock contention detection in multithreaded applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 253–262, New York, NY, USA, 2012. ACM.

[22] Intel. Using the rdtsc instruction for performance monitoring. `https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf`, 1997.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[24] Piotr Zalewski Jinwoo Hwang. Ibm thread and monitor dump analyze for java. `https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=2245aa39-fa5c-4475-b891-14c205f7333c`.

[25] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 217–228, New York, NY, USA, 2008. ACM.

[26] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 219–230, Berkeley, CA, USA, 2014. USENIX Association.

[27] 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.

[28] Xu Liu, John Mellor-Crummey, and Michael Fagan. A new approach for performance analysis of openmp programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 69–80, New York, NY, USA, 2013. ACM.

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

[30] Mecki. "when should one use a spinlock instead of mutex?". `http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex`, 2011.

[31] Wagner Meira, Jr., Thomas J. LeBlanc, and Alexandros Poulos. Waiting time analysis and performance visualization in carnival. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '96, pages 1–10, New York, NY, USA, 1996. ACM.

[32] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 106–113, New York, NY, USA, 1991. ACM.

[33] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 281–290, Washington, DC, USA, 2009. IEEE Computer Society.

[34] Notlikethat. "atomic operations on floats". `http://stackoverflow.com/questions/20981007/atomic-operations-on-floats`, 2014.

[35] Oracle. Hprof: A heap/cpu profiling tool. `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`.

[36] Oracle. Solaris Performance Analyzer. `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/o11-151-perf-analyzer-brief-1405338.pdf`.

[37] James Rapp. "diagnosing lock contention with the concurrency visualizer". `http://blogs.msdn.com/b/visualizeparallel/archive/2010/07/30/diagnosing-lock-contention-with-the-concurrency-visualizer.aspx`, 2010.

[38] Michael L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 31–40, New York, NY, USA, 2002. ACM.

[39] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 147–156, New York, NY, USA, 2010. ACM.

[40] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010.

[41] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[42] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou,

and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[43] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks, 2016.

[44] Long Zheng, Xiaofei Liao, Bingsheng He, Song Wu, and Hai Jin. On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 56–67, Washington, DC, USA, 2015. IEEE Computer Society.