# SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks

Tingting Yu
Department of Computer Science
University of Kentucky, USA
tyu@cs.uky.edu

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

## ABSTRACT

Writing concurrent programs is a challenge because developers must consider both functional correctness and performance requirements. Numerous program analyses and testing techniques have been proposed to detect functional faults, e.g., caused by incorrect synchronization. However, little work has been done to help developers address performance problems in concurrent programs, e.g., because of inefficient synchronization. This paper presents SyncProf, a concurrency-focused profiling approach that helps in detecting, localizing, and optimizing synchronization bottlenecks. In contrast to traditional profilers, SyncProf repeatedly executes a program with various inputs and summarizes the observed performance behavior. A key novelty is a graph-based representation of relations between critical sections, which is the basis for computing the performance impact of critical sections, for identifying the root cause of a bottleneck, and for suggesting optimization strategies to the developer. We evaluate SyncProf on 19 versions of eight C/C++ projects with both known and previously unknown synchronization bottlenecks. The results show that SyncProf effectively localizes the root causes of these bottlenecks with higher precision than a state of the art lock contention profiler and that it suggests valuable strategies to avoid the bottlenecks.

## CCS Concepts

•**Software and its engineering → Software notations and tools;**

## Keywords

Testing, Concurrency, Performance Bottlenecks

## 1. INTRODUCTION

Developing concurrent programs that are both correct and efficient is a challenge. On the one hand, a program must carefully synchronize concurrent accesses to shared data to avoid concurrency bugs, such as data races and atomicity violations. On the other hand, the program should avoid unnecessary or overly conservative synchronization because each synchronization operation may degrade the performance. Since these two goals, correctness and performance, are often contradictory, developers struggle to achieve both. A recent study reports that more than 25% of all critical sections (CSs) are changed at some point by the developers, both to fix correctness bugs and to enhance performance [22]. Another study shows that unnecessary synchronization is a common root cause for real-world performance problems [26]. Over the past decades, most research has focused on analyzing and debugging the correctness of concurrent programs, e.g., through detecting data races [7, 16, 29, 34] or atomicity violations [4,18,54,60,65], schedule exploration [8, 11, 38, 52, 53, 58], test generation [40, 44], and static analysis [19, 28, 39, 59, 63]. In contrast, the problem of detecting and avoiding concurrency-related performance problems is currently understudied.

For example, consider a performance problem in the `libvirt` KVM/QEMU driver [31]. A user reports a slowdown in virtual machine creation when multiple virtual machines are created in parallel. Figure 1a shows an excerpt of the execution of the `libvirt` program with five threads (T1 – T5) that contend for two locks (L1 and L2) while executing six CSs (CS1 – CS6). Each lightgray and darkgray horizontal box represents the execution of a CS, where the colors indicate which lock a thread must acquire to enter a CS. We illustrate the time a thread waits until obtaining the lock with a dotted line. After careful manual inspection of the code, it turns out that CS1 is unnecessarily synchronized with the other CSs that acquire lock L1. The developers fix the problem by splitting the lock L1 into two locks.

We call a performance problem due to unnecessary or inefficient synchronization a *synchronization bottleneck*. Such bottlenecks occur when multiple threads contend to reach a synchronization point, such as a lock acquisition, or when a single or multiple threads need to reach a synchronization point before other threads can make progress. The *root cause* of a synchronization bottleneck is the CS that causes the bottleneck and that needs to be changed to avoid it.

Unfortunately, synchronization bottlenecks are difficult to detect, understand, and address for several reasons. First, a bottleneck may not manifest with a particular test case and in a particular execution, e.g., because the workload does not expose the problem or because it is amortized by the rest of the execution. As developers cannot be expected to pick the "right" input and execution to reveal synchronization bottlenecks, one important challenge is to deal with a multitude of inputs and executions. Second, when a performance problem

*ISSTA'16*, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931070

manifests, it is hard to isolate synchronization bottlenecks from other problems, such as intensive I/O. Third, even if a problem is believed to be a synchronization bottleneck, it is non-trivial to locate its root cause. Complex multi-threaded programs may contain several dozens of CSs that involve dozens of locks, making it difficult to check all of them manually. Finally, not all synchronization-related performance problems can be easily optimized. For example, a CS may be needed because the program must synchronize concurrent accesses to shared data. To identify bottlenecks that can be optimized, developers must carefully reason about the behavior of all expensive CSs and filter those that can be modified without breaking the semantics of the program.

A promising way to address the above challenges is profiling, but existing approaches only partially address the problem. For example, Visual Studio's Concurrency Visualizer [10] provides a time profile that shows the time spent in different kinds of code segments, such as synchronization, I/O, and memory management. While effective at revealing symptoms of synchronization bottlenecks, such as heavy use of a lock, such approaches fail to link symptoms to their root causes. For the example in Figure 1a, a profiler based on idleness measurement reports CS5 as the most problematic CS, because it takes the longest time to be acquired. However, CS5 is not the root cause, and optimizing it does not improve performance. A profiler that measures the time spent inside each CS reports CS2 is the most expensive CS, because it is the CS in which the most time is spent. Again, optimizing CS2 does not improve the performance, as CS3 still must wait for CS1 to finish. Another problem is that profiling is based on individual executions. Manually examining the profiles of multiple executions is not an effective way to understand the overall performance because each profile may show different results for the same code segment. Finally, existing profiling approaches quantify the cost of synchronization but do not suggest possible optimizations.

This paper presents SyncProf, a profiler that detects synchronization bottlenecks, pinpoints their root cause, and suggests potential optimization strategies to address them. Given a program and tests to execute it, the approach dynamically analyzes multiple executions of the program and selects inputs where an increased workload increases the execution time while decreasing the CPU utilization. A key component of the approach is a novel graph representation of wait relationships between CSs. SyncProf generates a graph from executions and computes several metrics that summarize the impact of individual CSs on the overall execution time. Finally, SyncProf reports a ranked list of likely root causes of synchronization bottlenecks, along with their descriptions (e.g., code locations). Furthermore, SyncProf identifies instances of known bottleneck patterns that are amenable to particular optimizations, such as removing an unnecessary lock or splitting a lock, and suggests a specific optimization strategy for reported bottlenecks. The presented approach provides several benefits:

- SyncProf considers multiple inputs and executions, automatically selects those that are likely to expose a concurrency-related performance problem, and summarizes them into a set of synchronization bottlenecks.
- SyncProf isolates synchronization-related performance problems from other kinds of performance problems, providing concurrency-savvy developers a technique to address these particularly intricate issues.
- SyncProf pinpoints the root cause of bottlenecks, relieving the developer from manually reasoning about the interactions between multiple locks and CSs. Instead, developers can focus on those CSs that will benefit most from optimization.
- SyncProf simplifies the task of optimizing bottlenecks by suggesting bottleneck-specific optimization strategies. The approach does not fully automatically improve the performance, but it leaves the final decision if and how exactly to address a synchronization bottleneck to the developer.

We envision the approach to be used in at least two usage scenarios. First, a developer that is not aware of any concurrency bottlenecks in a program may profile the program with SyncProf to check if any such bottleneck exists. Second, a developer that knows that a program suffers from a concurrency-related bottleneck can use SyncProf to localize the problem. These usage scenarios are similar to traditional testing and fault localization, respectively, but for performance problems.

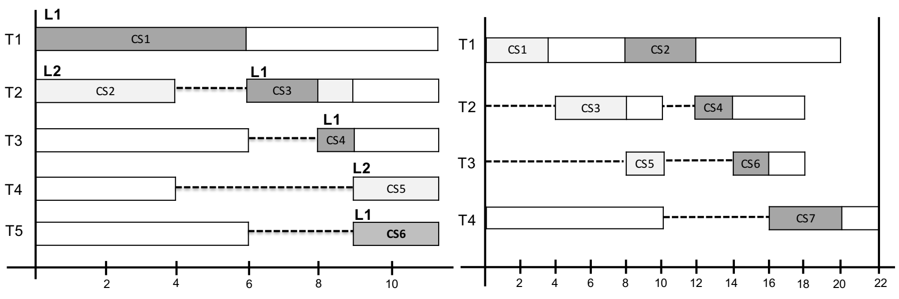
To evaluate the effectiveness of SyncProf, we apply the approach to popular benchmarks and real-world C/C++ programs with both known and previously unknown synchronization bottlenecks. Our results show that SyncProf effectively identifies the root causes of these bottlenecks and suggests profitable optimization strategies. Compared to a state of the art lock contention profiler, Valgrind's DRD tool, SyncProf pinpoints the root cause of a bottleneck with higher precision: SyncProf summarizes many executions and requires the developer to inspect between 1 and 3 CSs, whereas DRD requires the developer to inspect various execution profiles, most of which rank the root cause at a significantly lower rank than SyncProf. Addressing the bottlenecks pinpointed by our approach yields performance improvements between 17% and 160%.

In summary, this paper contributes the following:

- A concurrency-focused profiler that detects synchronization bottlenecks, pinpoints their root cause, and suggests bottleneck-specific optimization strategies.
- A novel graph representation of interactions between CSs. The graphs provide a generic basis for computing metrics that summarize the performance impact of individual CSs on the overall execution time and for suggesting synchronization bottleneck-specific optimization strategies.
- A practical implementation and empirical evidence that the approach effectively pinpoints performance problems in real-world C/C++ programs.

## 2. APPROACH

This section presents SyncProf, a profiling approach that helps developers to identify, localize, and optimize synchronization bottlenecks. Figure 2 gives an overview of the approach. The input to the approach is a program and test cases to exercise the program. Each test case is assumed to have a parameter to increase its workload, e.g., by increasing the input size or the number of threads. The approach consists of four parts. First, SyncProf executes the program's tests and identifies tests and workload sizes that expose a synchronization bottleneck. Second, the approach measures


(a) A real-world example

(b) Another example

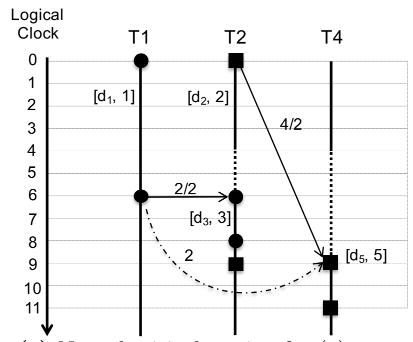

(c) Nested critical section for (a).

**Figure 1: Examples of synchronization bottlenecks and a partial synchronization dependence graph for (a).**

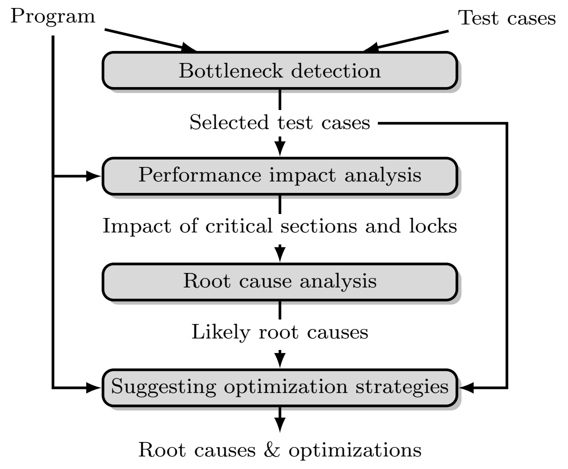

Program
Test cases
Bottleneck detection
Selected test cases
Performance impact analysis
Impact of critical sections and locks
Root cause analysis
Likely root causes
Suggesting optimization strategies
Root causes & optimizations

**Figure 2: Overview of the SyncProf approach.**

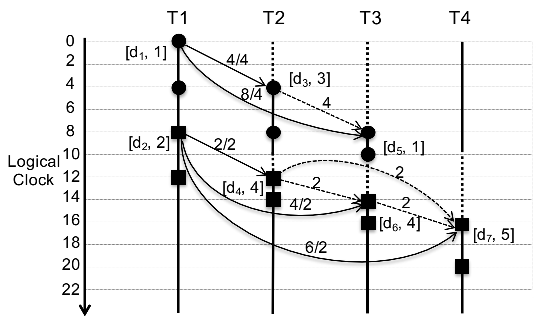


**Figure 3: Synchronization dependence graph for Figure 1b.**

the performance impact of CSs and summarizes the impact of individual CSs across multiple inputs and executions into a graph. Third, SyncProf uses this summary to identify bottlenecks and their likely root causes. Finally, the approach matches each identified bottleneck against common bottleneck patterns and, if the matching is successful, suggests an optimization strategy that is likely to address the performance problem. The output of SyncProf is a ranked list of CSs that are the likely root cause of a synchronization bottleneck, along with a suggested optimization strategy for some of the CSs.

## 2.1 Bottleneck Detection

The first part of SyncProf identifies test cases that expose synchronization bottlenecks. A test case $T_w$ has a program-specific parameter $w$ that specifies the size of the workload. For example, in a file compression program, possible workload parameters are the file size and the number of threads; in a server-side web application, a typical workload parameter is the number of requests per time [3,13,37]; in a database system, workload parameters may include a variety of attributes, such as the number of database queries per time, the number of tables, and the number of threads [5].

Given a set of parametrized test cases, SyncProf executes each test case with an increasing workload size. Identifying synchronization bottlenecks based on these executions is non-trivial because an increasing execution time is not a sufficient criterion. One reason is that increasing the workload size is expected to increase the execution time. Another reason is that synchronization bottlenecks may be hidden among other time-consuming behavior, especially among expensive but necessary behavior [66], such as I/O-intensive operations.

To address this challenge, SyncProf analyzes two symptoms of suboptimal performance: execution time and CPU usage. The approach identifies test cases where increasing the workload size leads to an increased execution time while the CPU usage is below a configurable threshold (default: 90%). We perform the following steps for each test case. At first, the approach executes the test case $N$ times (default: $N = 20$) and obtains a set of execution times $\mathcal{M} = \{t_1, t_2, \ldots, t_N\}$ and CPU usage values $\mathcal{U} = \{u_1, u_2, \ldots, u_N\}$. Repeating test execution is necessary to deal with the non-deterministic performance behavior of concurrent programs. Then, SyncProf increases the workload size for the test case and again executes the test case $N$ times, giving execution times $\mathcal{M}'$ and CPU usage values $\mathcal{U}'$. Now, the approach checks whether the two required symptoms manifest. To this end, SyncProf performs statistical analysis that check (1) whether the execution time in $\mathcal{M}'$ are significantly larger than the times in $\mathcal{M}$, (2) whether the values in $\mathcal{U}'$ are not significantly larger than the values in $\mathcal{U}$, and (3) whether the mean of $\mathcal{U}'$ is less than 90%. If these conditions hold, then SyncProf keeps the test case and the workload size for the remaining parts of the approach. Otherwise, the approach continues to increase the workload size. The increments of the workload size are provided along with the tests. For example, for a database system, one may increase the table size by 100 in each round. If the symptoms do not manifest after $\delta_{max}$ rounds (default: $\delta_{max} = 20$), the approach discards the test case.

An alternative to our approach would be to identify bottlenecks by measuring the percentage of time spent on synchronization. However, this alternative approach requires to instrument the program, imposing a significant runtime.

Instead, our approach is lightweight, enabling the first part of SyncProf to consider many tests and workload sizes.

## 2.2 Performance Impact Analysis

Given a set of test cases that are likely to expose synchronization bottlenecks, the second part of SyncProf computes the performance impact of each CS or lock on the overall execution time. The performance impact analysis builds upon a graph representation that summarizes the relations of CSs during a particular execution. To obtain this graph, SyncProf instruments the program and executes the test cases identified in the first part of the approach.

### 2.2.1 Synchronization Dependence Graph

SyncProf summarizes the synchronization-related behavior of a test execution into a graph:

Definition 1. *A synchronization dependence graph is a graph* $(V, E)$. *V is a set of nodes* $\{d_1, d_2, \dots d_i\}$, *where* $d_i$ *is a dynamic instance of a CS. E is a set of directed causality edges* $E = \{d_i \rightarrow d_j\}$, *implying* $d_j$ *is waiting for* $d_i$.

To uniquely identify CSs, SyncProf statically computes an identifier for each CS. The identifier is based on the entry and exit instructions of a CS, such as `pthread_mutex_lock` and `pthread_mutex_unlock`. Furthermore, to distinguish different paths through the CS, SyncProf computes a separate identifier for each acyclic control-flow path between the entry and exit instructions.

A SyDG is a connected or disconnected graph that consists of one or more *connected* subgraphs. Each subgraph depicts the dependence among dynamic instances of CSs.

The causality edges are constructed in three categories. A *direct waiting* edge reflects that a CS attempts to obtain the lock that is currently held by another CS. An *indirect waiting* edge reflects that $d_i$, which originally waits for $d_j$, now waits for a new dynamic CS $d_k$. A *nested waiting* edge reflects that $d_i$, which waits for $d_j$, may also wait for another dynamic CS $d_k$ if $d_k$ is waited for by a CS nested within $d_j$.

Figure 3 displays a SyDG with two subgraphs from the example of Figure 1b. Each node [$d_i$, k] represents a dynamic instance $d_i$ of a static CS identifier $k$. For example, nodes [$d_1$, 1] and [$d_5$, 1] are two dynamic instances of CS1. The oval and rectangle shapes indicate two different lock objects. The vertical dotted line in a thread indicates the time this thread spent waiting for a CS. The solid lines reflect direct waiting edges and the dotted lines reflect indirect waiting edges between two CS instances. For example, [$d_3$, 3] is directly waiting on [$d_1$, 1] and [$d_5$, 1] is indirectly waiting on [$d_3$, 3]. Each edge is assigned to a cost value (the edge labels are described later in the section). It is possible that a thread has to wait for one CS while it is executing another CS (i.e., nested CSs). Figure 1c displays a partial SyDG for Figure 1a that involves nested CSs, where the dash-dotted line indicates a nested waiting edge. Since $d_2$ is the outer CS of $d_3$, a nested waiting edge (the dashed line) is added from $d_1$ to the waiter of $d_2$ (i.e., $d_5$).

### 2.2.2 Constructing the Graph

SyncProf constructs the SyDG for an execution by instrumenting the program and by analyzing the execution trace. The trace is a sequence if synchronization events, i.e., acquiring a lock, obtaining a lock, and releasing a lock. Algorithm 4 summarizes the interpretation of the execution trace. A CS is set to be active on the current thread upon obtaining a lock (line 10) and inactive upon releasing a lock (line 22). When a `lock acquire` event is encountered, the algorithm adds into the SyDG a node $d$, which represents the dynamic instance of a CS associated with the current trace event (line 5).

```
procedure BuildSyDG (trace)
1:  V = $\phi$ ; E = $\phi$
2:  for each synchronization event $e$ in $trace$
3:      switch $e$
4:          case lock acquire:
5:              V.addNode($d$)
6:              if $d$ is blocked by $d'$
7:                  E.addEdge($d'$, $d$)
8:                  c($d'$,$d$) $\leftarrow$ $d.t$
9:          case lock obtain:
10:             setActive ($d.tid$) $\leftarrow$ $true$
11:             for each $d' \in$ in($d$)
12:                 c($d$, $d'$) $\leftarrow$ c($d$, $d'$) - $d.t$
13:                 for each $d'' \in$ (out($d'$)/($d$, $d'$))
14:                     E.addEdge($d$, $d''$)
15:                     c($d$, $d''$) $\leftarrow$ $d.t$
16:                     c($d'$, $d''$) $\leftarrow$ $d.t$ - c($d'$, $d''$)
17:                 for each $o \in$ active($d.tid$)/$d$
18:                     for each $d'' \in$ out($o$)
19:                         E.addEdge($d'$, $d''$)
20:                         c($d'$,$d''$) $\leftarrow$ c($d'$, $d$)
21:         case lock release:
22:             setActive ($d.tid$) $\leftarrow$ $false$
```

**Figure 4: Algorithm to compute SyDG.**

A direct waiting edge is added when a CS $d'$ attempts to obtain the lock which is held by a CS $d$ (line 7). The cost of the edge $c(d, d')$ is the time $d'$ that spends on waiting for $d$. The $c(d, d')$ is temporarily set to the current time (line 8) until $d'$ obtains the lock (line 9). When $d$ obtains the lock, the algorithm iterates through in-going edges of $d$ and finds out the CSs that $d$ was previously waiting on (i.e., waiters of $d$); the cost of each edge is updated by subtracting the current time from the time when $d$ started waiting (line 12). In the example of Figure 3, when $d_5$ is entered, the cost of edge ($d_1$, $d_5$) is 8 (shown on the left of the slash).

Next, the algorithm iterates through each CS $d''$ that was previously waiting on $d'$ (line 13). An indirect waiting edge is added from $d$ to $d''$ (line 14), indicating that $d''$ begins waiting on $d$ instead of $d'$. As such, the cost of ($d$, $d''$) is temporarily set to the current time (line 15), and then updated when $d''$ is entered (line 12). In the meantime, direct waiting edge ($d'$, $d''$) is updated by subtracting the current time from the old edge cost (line 16). In the example of Figure 3, when $d_3$ is entered, an indirect waiting edge (the dotted line) is added from $d_3$ to $d_5$, indicating that $d_5$ becomes a waiter of both $d_1$ and $d_3$. The cost of a direct waiting edge ($d_1$, $d_5$) is then reduced from 8 to 4 (shown on the right of the slash).

To construct SyDGs in the case of nested CSs, the algorithm first iterates through all outer CSs of $d$ by locating each active CS $o$ (excluding $d$) on the current thread (line 17). The waiters of $o$ are in turn indirectly waiting on $d'$. In Figure 1c, when $d_3$ is entered, the cost of ($d_1$, $d_3$) is updated to 2. The cost of the nested waiting edge is equal to c($d_1$, $d_3$). In the meantime, the cost of edge ($d_2$, $d_5$) is reduced to to $4 - 2 = 2$. Thus, the wait time incurred by T4 is attributed to both CS1 and CS2.

The timestamps used to construct SyDGs are based on a logical clock, which measures the number of executed conditionals (i.e., direct/indirect calls and direct/indirect branches) instead of actual wall-clock execution time. The rational for measuring time through this proxy metric, which is inspired by [46], is twofold. First, accurately measuring the time span between two events that are close to each other in time is challenging due to the limited precision of timestamps provided by the operating system. As a result, measuring wall-clock time risks to yield inaccurate and potentially misleading values. Second, as the instrumentation added by SyncProf influences the execution time of the profiled program, measurements of wall-clock execution time may be distorted. In contrast, the number of executed conditionals is not influenced by instrumentation.

### 2.2.3 *Performance Impact Metrics*

Based on the SyDG of an execution, SyncProf quantifies the performance impact of CSs and locks. The result of this step is a set of impact values for each test $T$, denoted by $PI_T$. We use $PI_{T,CS}$ to denote the cost value associated with a particular CS, where $CS$ refers to the unique static CS. Unlike existing contention measurement approaches [9, 27, 56] that focus on a specific metric, SyDGs provide a generic representation that enables SyncProf to compute multiple metrics that summarize the performance impact of CSs. Here, we introduce three metrics used by SyncProf.

**All-path Wait Time (APWT).** This metric measures the performance impact of a CS by aggregating the time spent by *all* other threads waiting for the CS:

$$PI_{T,CS} = \sum_{(v \in V(SyDG_T) \wedge v.sid = CS)} \sum_{e \in out(v)} c(e)$$

$V(SyDG_T)$ are all nodes in $SyDG_T$ obtained by exercising test $T$, $v.sid$ is the CS identifier, $out(v)$ are the outgoing edges of $v$, and $c(e)$ is the cost of an outgoing edge of $v$.

For example, consider the SyDG constructed for the example in Figure 3. The performance impact of CS1 is the sum of performance impacts across all dynamic instances (i.e., [$d_1$,1] and [$d_5$,1]) of CS1, that is, $PI_{T,1} = 4 + 4 = 8$, Likewise, $PI_{T,2} = 2 + 2 + 2 = 6$.

**Critical-path Wait Time (CPWT).** Considering wait time can be effective at identifying synchronization bottlenecks, the results may be misleading when the synchronization bottleneck does not impact the completion time of a program. To address this problem, the CPWD metric considers the critical path[1] [6, 9] of an execution while quantifying the performance impact of a CS. Applying CPWT requires isolating critical path graphs from other subgraphs in each SyDG:

Definition 3: *A critical path graph is a subgraph of a SyDG, where the last node of each connected component is from the last finished thread.*

In the example in Figure 3, since T4 is the last finished thread, the critical path graph contains the nodes $d_2$, $d_4$, $d_6$, and $d_7$. In contrast to APWT, CPWT concludes that CS2 induces the highest performance impact.

SyncProf also enables developers to combine multiple metrics. In this case, the performance impact of a CS is the mean of the impacts computed by multiple metrics.

**All-path Lock Time (APLT).** The APWT and CPWT metrics quantify performance impact for individual CSs. In some cases, optimization is done for multiple CSs associated with the same lock. The APLT metric enables developers to analyze the performance impact for individual locks. The APLT measures the total time spent by other threads waiting for $L$, denoted by $PI_{T,L}$. Specifically, for each lock object $L$, SyncProf locates the nodes in a SyDG associated with $L$, and adds up the costs of all their outgoing edges.

In the example in Figure 3, where the two SyDGs involve different locks $L1$ (top subgraph) and $L2$ (bottom subgraph). Thus, $PI_{T,L1} = 4 + 4 + 4 = 12$, and $PI_{T,L2} = 2 + 2 + 2 + 2 + 2 + 2 = 12$.

### 2.2.4 *Dealing with Non-Deterministic Performance*

Different program executions for one input may expose different performance properties due to the non-deterministic behaviors of multi-threaded programs. To mitigate such non-determinism, SyncProf takes additional executions for each input and considers only performance impact values that vary within specified bounds. Specifically, SyncProf assumes a fixed testing budget $B$ for executing all test cases. Let $N$ be the number of times one can repeat a test case $T$ within $B$. SyncProf repeatedly executes test $T$ and calculates $PI_{T,CS}$ for each CS (see Section 2.2.3). At the end of each repetition, SyncProf accumulates $PI_{T,CS}$ using a set $M_{T,CS}$, i.e., $M_{T,CS} = \{PI_{T,CS_1},\ PI_{T,CS_2},\ \dots,\ PI_{T,CS_n}\}$, where $n$ is the $n^{th}$ repetition and $n \leq N$. Next, the standard deviation of $M_{T,CS}$ is calculated, denoted by $\sigma(PI_{T,CS})$. SyncProfstops repeating executions when $\sigma(PI_{T,CS})$ is below a specified threshold $\delta_{stop}$ or when $n$ reaches the maximum value $N$. By default, SyncProf sets $\delta_{stop}$ to the percentage ($\sigma_p$) of the mean of the $M_n$, that is, $\overline{M_{T,CS}} \cdot \sigma_p$ [45]. For the evaluation, we use a testing budget $B$ of 12 hours, $\delta_{stop} = 0.01$, and $N = 10$.

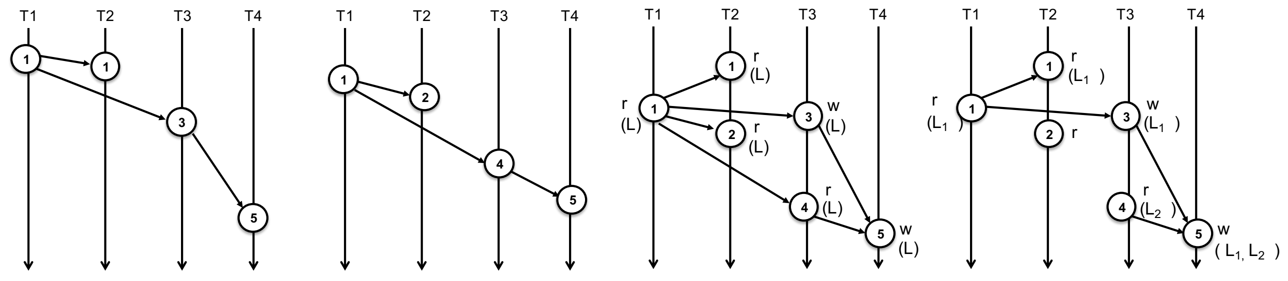
It is possible that the standard deviation $\sigma(PI_{T,CS})$ is above the specified threshold $\delta_{stop}$ after the number of repetitions reach $N$. In this case, SyncProf will report the CS $CS$ as inconclusive [45] and leave it for manual inspection. However, we did not find any inconclusive test cases in our study. The final PI$_{T,CS}$ is the mean of $M_{T,CS}$, excluding any inconclusive test cases, i.e., PI$_{T,CS}$ = $\overline{M_{T,CS}}$.

## 2.3 Root Cause Analysis

Given the performance impact values of all critical sections and tests, SyncProf ranks CSs by their likelihood to be the root cause of a bottleneck. Each CS corresponds to a set $PI_{CS} = \{\overline{M_{1,CS}}, \overline{M_{2,CS}}, \dots, \overline{M_{T,CS}}\}$ of performance impact values, where $T$ is the index of a test input. Next, SyncProf ranks the CSs in terms of the mean of each $PI_{CS}$. Formally, SyncProf considers $CS_1$ to be more critical than $CS_2$ if $\overline{PI_{CS_1}} > \overline{PI_{CS_2}}$. Once CSs are prioritized, SyncProf enables developers to choose which CSs to optimize and suggests optimization strategies.

Another possible approach is to perform a pairwise analysis between all CSs and to conclude that one CS is more performance impact than another when there is a statistically significant difference. We rejected this idea for two reasons. First, this approach ranks CSs in a partial order, which could be difficult for developers to understand. Second, conducting multiple hypothesis tests might trigger the multiple comparison problem and therefore compromise the validity of the ranking.

[1]A path that directly impacts the completion time of a program.



**(a)** SyDG$_1$ **(b)** SyDG$_2$ **(c)** USyDG **(d)** Updated USyDG

**Figure 5: Optimization example.**

**Table 1: Example of suggested optimization.**

| *CS ID* | *lock* | *object* | *source location* | *action* |
|---|---|---|---|---|
| 1 | `mutex_enter` | `g_lock` | `foo.c`: (1, 10) | -> L1 |
| 3 | `mutex_enter` | `g_lock` | `foo.c`: (21, 29) | -> L2 |
| 5 | `mutex_enter` | `g_lock` | `bar.c`: (2, 11) | -> Remove |
| 6 | ... | ... | ... | ... |

## 2.4 Suggesting Optimization Strategies

The final step of SyncProf heuristically suggests optimization strategies for the detected synchronization bottlenecks. We consider three optimizations: *unnecessary synchronization elimination*, *lock split*, and *reader-writer locks*. SyncProf reports descriptions of suggested optimizations, that each contain a static CS identifier, locks and objects, code locations, and suggested actions. Table 1 describes a sample output, which indicates that CS1, CS3 and CS5 are protected by the same lock (i.e., `mutex_enter` with object `g_lock`). This lock can be split into two locks on CS1 and CS3, and can be removed on CS5.

To obtain such suggestions, SyncProf performs the following steps. First, the approach instruments synchronization operations, CS entry and exit points, and memory reads and writes. The tests are exercised against the instrumented program to generate a new set of SyDGs specific for optimization (denoted as SyDG$_o$). The SyDG$_o$ is constructed in the same way as the original SyDG (Section 2.2), except that a SyDG$_o$ does not record or update timestamps. Every SyDG$_o$ node maintains a read set ($CS_{rd}$) and a write set( $CS_{wt}$) that record all memory locations accessed in the CS. To mitigate runtime overhead, SyncProf instruments only the top $K$ CSs in the ranking and the CSs that use the same lock as the $K$ CSs; this information can be retrieved by analyzing the original SyDGs. The above process of generating SyDG$_o$s is repeated $N$ times. Thus, the number of traces is $|T| * N$, where $|T|$ is the number of tests. We set $K = 5$ and $N = 20$ in the evaluation.

Next, SyncProf transforms all SyDG$_o$s into a *universal synchronization dependence graph* (*USyDG*). To do this, for each thread across all SyDG$_o$s, the nodes with common static CS identifiers are merged on this thread, and their read and write sets are joined, respectively. The incoming or outgoing points of the associated edges are merged. Figure 5a and Figure 5b display two SyDGs from two tests $t_1$ and $t_2$. Figure 5c displays the USyDG by merging $SyDG_1$ and $SyDG_2$, where the two CSs (node 1) in T1 are merged into one node. In the USyDG, $r$ indicates there exists a shared variable read in a node (i.e., $CS_{rd}$ = {r}), and $w$ indicates the same shared variable is written (i.e., $CS_{rd}$ = {r}). The ($L$) indicates a CS is protected by a lock $L$.

*Identifying Optimization Patterns.* To identify optimizable CSs, SyncProf considers three patterns to match with each pair of connected nodes ($m$ and $n$) in the USyDG: (1) *null-shared*, (2) *read-read*, and (3) *low-degree-conflicts*.

The *null-shared* pattern happens when there exist no shared memory accesses between two CSs. Two USyDG nodes match the *null-shared* pattern if $((CS_{rd_m} \cup CS_{wt_m}) \cap (CS_{rd_n} \cup CS_{wt_n})) = \phi$. The *read-read* pattern refers to the case where two CSs protected by the same lock access the shared variable but none of them is a write. This pattern is matched if $(CS_{rd_m} \cap CS_{wt_n} = \phi) \wedge (CS_{rd_n} \cup CS_{rd_n} \neq \phi)$.

*Unnecessary Synchronization Elimination.* If a pair of nodes matches either pattern (1) or (2), and if the static CS identifiers in the pair are not identical, the edge for the pair is removed. In the example of Figure 5c, edges (T1:1,T2:2), and (T1:1,T3:4) are removed (updated in Figure 5d), because they both match the *read-read* pattern. The edge (T1:1,T2:1) is not removed because their node identifiers are identical. Next, for each standalone node in the updated USyDG, SyncProf suggests to remove the synchronization in the node, as the node does not have dependences on other CS nodes. In Figure 5d, node 2 in T2 is removed.

*Lock Split.* For the remaining connected USyDG nodes, SyncProf reconstructs lock dependences to suggest fine-grained locks that guard disjoint sets of shared variables. To do this, SyncProf first removes the original locks in each node, and then uses dummy locks to reconstruct its lock set. Specifically, SyncProf assigns a dummy lock to every node $s$ that has only outgoing edges. A node $t$ with incoming edges should be synchronized by the given lock of its source node $s$. Thus, the lock set of $t$ is updated by joining with the lock set of $s$. In the meantime, the lock sets of all other nodes with the same identifiers are updated to ensure consistency. In the example of Figure 5d, node 1 (on T1) is assigned a new dummy lock L$_1$ and node 4 is assigned a dummy lock L$_2$. The lock set of node 1 on T2 is also updated to L$_1$. Next, node 3 and node 5 are updated by joining the lock set of node 1 (i.e., L$_1$). In the end, the original lock $L$ is split into two locks so that node 4 does not acquire the same lock as node 1.

*Reader-writer Locks.* SyncProf can further suggest reader-writer locks. To do this, for each node $s$ with a non-empty write set, SyncProf finds all undirected simple paths starting from $s$. If all nodes except $s$ in a path have common node identifiers, and if the write sets of these nodes are all empty, the node identifier is a reader of $s$. In Figure 5d, <3:T3, 1:T1, 1:T2> is a simple path starting from node 3.

In fact, node 1 is a reader of node 3 and can be executed concurrently. Thus, node 1 and node 3 can be protected by a reader-writer lock.

The *low-degree-conflicts* is a special pattern that indicates that two CSs protected by the same lock both read and write to a shared memory, but such conflicting accesses occur very infrequently. To identify the degree of conflicts between two CSs, SyncProf measures the percentage of two consecutive executions of a CS pair that matches either pattern (1) or pattern (2) (i.e., a non-conflicting access pattern) over all consecutive executions of the pair. If the percentage exceeds a threshold $\delta_{deg}$, the CS pair matches the *low-degree-write* pattern. We set $\delta_{deg} = 80\%$ as a default.

In practice, it is almost impossible to suggest concrete optimizations for the *low-degree-write* pattern, as it often depends on program design and thus requires developers' knowledge to optimize the code. For example, developers may choose to reduce the size of CSs, set flags to enable synchronizations under certain condition, applying specific data structures (e.g., non-blocking algorithm [35]). In this case, SyncProf only reports CSs that match pattern (3), and leaves them for developers to investigate.

## 3. IMPLEMENTATION

SyncProf is implemented on top of the PIN instrumentation framework [33]. To obtain CS identifiers, we use CodeSurfer[2] to perform context-sensitive and flow-sensitive analysis that enumerates paths enclosed in each CS. The instrumentation considers entry and return instructions of all synchronization operations, event types, identifiers of synchronization objects, and identifiers of thread and memory accesses. All these events are recorded using APIs provided by PIN. When a test run completes, a SyDG along with its runtime information is recorded into a trace file, which is fed into the analysis modules for further processing.

## 4. EVALUATION

We apply SyncProf to several C/C++ programs to address four research questions:

**RQ1:** How effective is SyncProf at identifying synchronization bottlenecks?

**RQ2:** How does SyncProf compare to a state-of-art profiler?

**RQ3:** How effective is SyncProf at suggesting profitable optimizations?

**RQ4:** How efficient is SyncProf?

### 4.1 Experimental Setup

*Benchmark Programs.* Table 2 lists the C/C++ programs we use in the evaluation. We use programs with known bottlenecks to evaluate whether SyncProf identifies them and suggests profitable optimizations. We also use the latest program versions to evaluate whether SyncProf detects previously unknown synchronization bottlenecks.

The first six programs are benchmark programs used by others to study concurrency and performance [9, 25, 42, 43, 49, 64]. Only the first four benchmark programs have previously known bottlenecks [9, 49, 64]. The remaining programs are different versions of popular open source projects. To obtain known bottlenecks, we searched the project's issue trackers for synchronization-related problems and randomly

**Table 2: Benchmark programs.**

| *Program* | *NLOC* | *Issue* | *T* | $T_{sel}$ | *CS* | $CS_{cov}$ | *OBJ* | $OBJ_{cov}$ |
|---|---|---|---|---|---|---|---|---|
| UTS | 4.9K | [9] | 16 | 8 | 17 | 17 [8, 14] | 6 | 6 [6, 6] |
| Radio. | 8.2K | [9] | 22 | 14 | 54 | 54 [21, 32] | 6 | 6 [6, 6] |
| Ocean | 2.6K | [24] | 44 | 21 | 44 | 44 [25, 30] | 15 | 15 [13,15] |
| Barnes | 2.0K | [64] | 12 | 9 | 12 | 12 [9, 12] | 4 | 4 [4, 4] |
| Cholesky∗ | 3.7k | - | 21 | 12 | 6 | 6 [5,6] | 4 | 4 [4, 4] |
| FMM∗ | 3.2k | - | 26 | 0 | 28 | 28 [21, 24] | 9 | 9 [9, 9] |
| MySQL1 | 199K | 38941 | 98 | 31 | 870 | 220 [52, 67] | 156 | 25 [17, 19] |
| MySQL2 | 236.9K | 62018 | 98 | 29 | 264 | 82 [29, 40] | 34 | 28 [21, 24] |
| MySQL3 | 315.5K | 73361 | 94 | 32 | 462 | 118 [35, 48] | 67 | 38 [24, 30] |
| MySQL4 | 413.7K | 75534 | 83 | 21 | 556 | 122 [30, 39] | 38 | 23 [17, 22] |
| MySQL5 | 398.0K | 72829 | 85 | 24 | 322 | 98 [25, 29] | 73 | 42 [29, 35] |
| MySQL6 | 422.8K | 76509 | 91 | 43 | 324 | 105 [32, 42] | 38 | 28 [20, 23] |
| MySQL7 | 427.8K | 76686 | 102 | 25 | 333 | 109 [28, 45] | 40 | 28 [20, 22] |
| MySQL8 | 315.5K | 77094 | 100 | 25 | 329 | 102 [32, 40] | 73 | 45 [29, 38] |
| MySQL9∗ | 443.1K | - | 112 | 28 | 336 | 114 [35, 48] | 76 | 49 [31, 39] |
| Firefox1 | 1,120K | 733277 | 42 | 15 | 87 | 45 [34, 39] | 23 | 16 [10, 13] |
| Firefox2 | 1,258K | 488148 | 40 | 11 | 88 | 40 [29, 36] | 31 | 15 [9, 13] |
| Firefox3 | 1,223K | 121523 | 39 | 12 | 82 | 41 [20, 32] | 28 | 15 [8, 12] |
| Firefox4∗ | 2,169K | - | 45 | 0 | 196 | 90 [39, 71] | 35 | 22 [15, 20] |

NLOC=#lines of code. Issue=known synchronization bottleneck, $T$ and $T_{sec}$=#(selected) test cases. $CS$ and $CS_{cov}$=#(covered) critical sections with 95%-confidence intervals. $OBJ$ and $OBJ_{cov}$=#(covered) synchronization objects with 95%-confidence intervals.

selected ten MySQL issues from the latest two major versions (i.e., 5.6.x and 5.7.x) and three Firefox issues. We could reproduce eleven issues in our environment (MySQL1 to MySQL8 and Firefox1 to Firefox3). In addition, we also include the latest versions of MySQL (5.7.10) and Firefox (44.0b9), listed as MySQL9 and Firefox4. Before applying SyncProf, we did not know any bottlenecks in these versions, indicated by ∗.

*Test Cases.* Table 2 gives the number of test cases for each program. For the benchmark programs, the workload can be specified as small, medium, or large. For MySQL, we use two existing test suites that trigger requests to the database: (i) mysqlslap, where the workload size is the number of queries, writes, indexes, and threads; (ii) sysbench, where the workload size is the number of tables, the table size, and the number of threads. For Firefox, we write Selenium tests that open multiple tabs with popular web sites simultaneously. The workload size is the number of requested sites.

*Previously Unknown, Optimizable Bottlenecks.* To evaluate whether SyncProf finds bottlenecks that are not among those known to us before running the profiler, we manually inspect reported code locations. If this inspection suggests that a bottleneck can be optimized, we check whether the code has been optimized by the developers in a later version of the program. If not, we patch the program as suggested by SyncProf and run its test cases. We consider an optimization is valid if it does not fail any test case and if it improves performance.

*Comparison with Valgrind Lock Contention Profiler.* To answer RQ2, we compare SyncProf to the Valgrind lock profiling tool DRD [14]. We choose DRD as a baseline because it is available as open source. DRD profiles individual test executions and does not select tests or summarize their behavior.

*Evaluating Effectiveness of SyncProf in Suggesting Optimizations.* To answer RQ3, we compare the optimization strategy suggested by SyncProf to the known optimiza-

[2]http://www.grammatech.com/products/codesurfer/overview.html

tions. We determine whether SyncProf's suggestion matches a known optimization by manually comparing the known optimization to the suggestion.

***Threats to Validity.*** The primary threat to external validity for this study involves the representativeness of our objects and test cases. Other objects and test cases may exhibit different behaviors and cost-benefit tradeoffs. However, we do reduce this threat to some extent by using several varieties of well studied open source code objects for our study, and test suites generated by practical approaches. Though generating performance test cases is not the focus of this work, it is true that different test cases may cause the programs to exhibit different behaviors.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

Where construct validity is concerned, it is true that optimizing bottlenecks can be subjective. To mitigate this threat, we used the developers' fixes for the reported issues as the ground truth to assess our approach.

## 4.2 Results and Analysis

### 4.2.1 RQ1: Effectiveness in Localizing Bottlenecks

To answer RQ1, we compare the ranked list of CSs reported by SyncProf with the critical section $CS_{opt}$ and the lock $L_{opt}$ that are improved in the known and previously unknown but beneficial optimizations. The higher the approach ranks $CS_{opt}$ and $L_{opt}$, the quicker the developer localizes the program location to optimize. While the size of CSs may also affect the manual effort required by developers, CSs are usually short. Based on a calculation on all programs, the CS size ranges from 2 to 52 lines of code, with an average of 10. Table 3 lists the bottlenecks identified by SyncProf (MySQL4-1 and MySQL4-2 indicate two bottlenecks found in MySQL4 (V 5.7.5)). FMM and Firefox4 are not listed because no synchronization bottlenecks are detected or have been known. The first block of columns shows the rank by the three metrics of SyncProf. The numbers in parentheses indicate the percentage of CSs (locks) that the developer would have to inspect among all executed CSs (locks) in the program.

SyncProf found two previously unknown and valid bottlenecks in Cholesky and MySQL9. We reported the bottleneck to the MySQL developers (bug #80101). SyncProf found another bottleneck that was previously unknown to us (MySQL4-2); it has been fixed independently of us in the next version of MySQL.

For all 18 detected bottlenecks, SyncProf guides the programmer to the desired CS by examining at most 5% of all CSs. In fact, for nine of 18 bottlenecks using the CPWT metric, the root cause is ranked *first* among all CSs. For four out of 18 bottlenecks, the CPWT metric is more effective than the APWT metric. For the other bottlenecks, the CSs that introduces significant performance impact are executed in short-execution threads. For 15 of 18 bottlenecks, the lock is ranked as the first among all locks using the APLT metric. We conclude that SyncProf is effective at pinpointing the root cause of synchronization bottlenecks and that the critical path metrics are particularly effective.

***Usefulness of Selecting Bottleneck-exposing Tests.*** We also evaluated whether the step of selecting bottleneck-exposing test cases impacts the effectiveness of SyncProf ("Rank by SyncProf w/o selection" of Table 3). Without this step, SyncProf was less effective for twelve out of 18 bottlenecks using the CPWT metric, showing that the test selection step is beneficial.

### 4.2.2 RQ2: Comparison with Existing Profiler

To compare SyncProf with the existing DRD profiler, we consider the ranked list of CSs that DRD reports as potential bottlenecks. In contrast to our approach, DRD does not summarize the profiling results of multiple test cases, leaving the task of picking the "right" test case to the developer. For a fair comparison, we run DRD on all test cases and compute the number $N$ of CSs to inspect before hitting the desired CS for each run. Columns "MAX" and "AVG" of Table 3 show the maximum and average number of CSs over all analyzed runs, respectively. The next column shows the confidence interval of $N$ over all analyzed runs, indicating the range in which $N$ is likely to be.

We compare SyncProf to DRD by comparing how many CSs a developer must inspect to find the root cause of a bottleneck. For SyncProf, we use a ranking that combines the performance impacts reported by both APWT and CPWT (column "AVG" in the first block). For DRD, we consider the average rank at which the root cause CS appears across all profiled executions. All rankings provided by SyncProf are strictly more effective in pinpointing the root cause than DRD. For example, for MySQL2, SyncProf reports the desired CS at rank one or three (depending on the metric), whereas DRD is likely to rank the CS at a position between 4.0 and 9.2, with an average rank of 6.8. To summarize the increase in precision of SyncProf over DRD, we compute for each program how much SyncProf reduced the number of CSs to inspect compared to DRD, and compute the geometric mean across all bottlenecks. Overall, SyncProf reduced the number of CSs to inspect by an average of 55%.

These results confirm two design decision of our approach. First, quantifying the wait time of CSs alone, as done by DRD, is not enough to evaluate the performance impact of CSs. Second, summarizing performance impact across test executions simplifies localizing bottlenecks.

### 4.2.3 RQ3: Effectiveness in Suggesting Optimizations

Table 4 lists the concrete optimizations and their detected patterns (1. *null-shared*, 2. *read-read*, and 3. *low-degree-conflicts*) separated by commas. The notation "-" indicates that a pattern is not found or a concrete optimization cannot be suggested. The "strikeout" line indicates a change that breaks the program's semantics. Column 3 lists the fixes for the known issues. The notation "✓" indicates the suggested optimization matches with the real fix. The last column lists the performance improvements after applying the optimizations. The performance improvement is calculated by averaging the improvements across all bottleneck-exposing test cases. To mitigate non-determinism, we ran each test case ten times.

As Table 4 shows, SyncProf finds optimization patterns in 14 out of 18 bottlenecks. For the 14 optimizable bottlenecks, SyncProf suggested nine concrete optimizations; seven of them matched the ground truth and were beneficial. Applying the optimizations led to performance improvements between 17% and 160%. We further explain these results in

Table 3: **Effectiveness and efficiency in localizing bottlenecks.**

| *Bottleneck* | *Rank by SyncProf (%)* | | | | *Time* | *Rank by SyncProf w/o selection* | | | *Time* | *Rank by DRD* | | | *Time* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *CPWT* | *APWT* | *APLT* | *AVG* | *(min)* | *CPWT* | *APWT* | *APLT* | *(min)* | *MAX* | *AVG* | *Conf. Int.* | *(min)* |
| UTS | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 49.5 | 2 (0.9%) | 2 (0.9%) | 2 (0.9%) | 86.1 | 3 | 1.3 | [1.2, 2.4] | 22.5 |
| Radio. | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 38.6 | 1 (0) | 1 (0) | 1 (0) | 64.3 | 3 | 1.2 | [1.1, 1.5] | 20.8 |
| Ocean | 1 (0) | 2 (2.3%) | 1 (0) | 1 (0) | 13.2 | 2 (2.3%) | 2 (2.3%) | 1 (0) | 85.2 | 4 | 2.5 | [1.8, 3.2] | 30.3 |
| Barnes. | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 44.5 | 1 (0) | 1 (0) | 1 (0) | 50.2 | 5 | 3.2 | [2.1, 3.6] | 19.3 |
| Cholesky∗ | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 36.0 | 2 (16.7%) | 2 (16.7%) | 2 (25%) | 52.2 | 3 | 1.3 | [1.2, 1.5] | 15.3 |
| MySQL1 | 2 (0.5%) | 2 (0.5%) | 1 (0) | 2 (0.5%) | 289.3 | 2 (0.5%) | 3 (0.9%) | 1 (0) | 783.2 | 7 | 3.6 | [2.9, 5.2] | 290.0 |
| MySQL2 | 1 (0) | 3 (2.4%) | 1 (0) | 3 (2.4%) | 288.4 | 2 (1.2%) | 3 (2.4%) | 2 (3.5%) | 892.2 | 11 | 6.8 | [4.0, 9.2] | 282.4 |
| MySQL3 | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 281.5 | 1 (0) | 1 (0) | 1 (0) | 832.2 | 5 | 1.5 | [1.1, 2.9] | 285.3 |
| MySQL4-1 | 2 (0.8%) | 2 (0.8%) | 1 (0) | 2 (0.8%) | 240.2 | 4 (2.5%) | 3 (1.6%) | 1 (0) | 450.4 | 11 | 4.3 | [2.9, 9.8] | 202.7 |
| MySQL4-2 | 5 (3.3%) | 5 (3.3%) | 3 (8.7%) | 5 (3.3%) | 240.2 | 6 (4.1%) | 7 (4.9%) | 4 (13%) | 450.4 | 11 | 8.1 | [6.4, 9.9] | 202.7 |
| MySQL5 | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 224.5 | 2 (1%) | 2 (1%) | 1 (0) | 800.3 | 5 | 2.9 | [2.5, 3.4] | 198.6 |
| MySQL6 | 2 (1%) | 3 (1.9%) | 1 (0) | 3 (1.9%) | 221.4 | 3 (1.9%) | 4 (2.9%) | 1 (0) | 462.5 | 12 | 5.2 | [3.7, 9.4] | 208.7 |
| MySQL7 | 2 (0.9%) | 2 (0.9%) | 1 (0) | 2 (0.9%) | 312.5 | 3 (1.8%) | 3 (1.8%) | 1 (0) | 1198.4 | 8 | 3.3 | [2.2, 5.3] | 300.0 |
| MySQL8 | 3 (2%) | 3 (2%) | 2 (2.2%) | 3 (2%) | 256.4 | 3 (2%) | 3 (2%) | 2 (2.2%) | 999.5 | 14 | 8.2 | [5.5, 11.8] | 228.4 |
| MySQL9∗ | 3 (1.8%) | 3 (1.8%) | 2 (2.2%) | 3 (1.8%) | 340.0 | 5 (3.5%) | 5 (3.5%) | 2 (2.2%) | 1198.4 | 14 | 8.2 | [5.5, 11.8] | 320.8 |
| Firefox1 | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 239.4 | 2 (2.2%) | 2 (2.2%) | 1 (0) | 618.5 | 6 | 3.0 | [1.8, 5.4] | 180.3 |
| Firefox2 | 2 (2.5%) | 3 (5%) | 1 (0) | 3 (5%) | 198.5 | 2 (2.5%) | 3 (5%) | 1 (0) | 600.3 | 13 | 9.1 | [6.4, 10.2] | 182.5 |
| Firefox3 | 3 (2.2%) | 3 (2.2%) | 1 (0) | 3 (2.2%) | 185.2 | 5 (4.4%) | 5 (4.4%) | 3 (9.1%) | 592.4 | 13 | 8.3 | [6.4, 10.2] | 169.3 |

Table 4: **Effectiveness in suggesting optimizations.**

| *Bottleneck* | *Suggested optimization* | *Ground truth* | *Imp.* |
|---|---|---|---|
| UTS | `-, (3)` | `nonblocking queue` | 17% |
| Radio. | `-, (3)` | `nonblocking queue` | 20% |
| Ocean | `-, -` | `shrinking critical sections` | 22% |
| Barnes. | `-, (3)` | `increase lock array` | 21% |
| Cholesky.∗ | `-, (3)` | `nonblocking queue` | 19% |
| MySQL1 | `-, -` | `replace a random generator` | 80% |
| MySQL2 | `lock split, (1,2)` | ✓ | 160% |
| MySQL3 | `lock elimination, (2)` | ✓ | 125% |
| MySQL4-1 | `lock split, (1,2)` | ✓ | 140% |
| MySQL4-2 | `use reader locks, (2)` | ✓ | 21% |
| MySQL5 | `lock split, (1,2)` | ✓ | 40% |
| MySQL6 | ~~locks elimination,(1,2)~~ | `set conditions` | 18% |
| MySQL7 | `-, (3)` | `partition accesses` | 21% |
| MySQL8 | `-, -` | `add an additional buffer` | 42% |
| MySQL9∗ | ~~locks elimination,(2)~~ | `set conditions` | 22% |
| Firefox1 | `lock split, (1,2)` | ✓ | 40% |
| Firefox2 | `lock split, (1,2)` | ✓ | 28% |
| Firefox3 | `-, (3)` | `partition accesses and CS split` | 95% |

four categories.

*Patterns Found and True Optimizations Suggested.* For seven bottlenecks, SyncProf finds optimization patterns and suggests optimizations that match the ground truth. For example, in MySQL3, when a dummy table is created, the CS protected by the `zip_pad.mutex` lock is a top synchronization bottleneck. SyncProf detected several *read-read* patterns for this CS, and suggests to remove the lock. Likewise, on MySQL4-2, the lock used in the `update_on_commit` is suggested to be replaced with a reader lock.

In MySQL2, MySQL4-1, MySQL5, Firefox1, and Firefox2, the optimizations involve splitting the locks for finer-grained locking. For example, in MySQL4-1, the buffer pool mutex that protects a number of CSs was suggested to split into four different mutex objects.

*Patterns Found and Optimizations Not Suggested.* For five bottlenecks, SyncProf detected optimization patterns involving *low-degree conflicts* but does not suggest optimizations targeting specific CSs. For example, on UTS, Radiosity and Cholesky, the shared queue is protected by a lock, but the conflict accesses rarely happen. This bottleneck can be optimized with a non-blocking queue algorithm. In Barnes, a lock array is too small to support fine-grained locking, which can be optimized by increasing the array index [64]. Suggesting concrete optimizations for the above bottlenecks requires a deep understanding of the semantics of the programs.

In MySQL7, the threads waiting for a condition variable are simultaneously waken up and contend to enter the same CS. This CS rarely involves conflicting accesses. The real fix is to use separate condition variables to wake up threads in multiple phases. In Firefox3, the optimization is to partition the session lock into a bucket of locks, and each lock is indexed in different sessions. Again, these optimizations require the developer's knowledge.

*Patterns Found and Optimization Falsely Suggested.* In MySQL6, a CS is used to examine a list of plugins. Since no plugins are installed in the tested program, SyncProf detected several *read-read* patterns in this CS. However, conflicting accesses may occur when plugins are installed. The real fix is to use counters to track the loading and unloading of each plugin and to enable the lock when the counter is not zero. Suggesting such optimizations requires additional software components.

In MySQL9, SyncProf detected several *read-read* patterns in the CS of the function `lock_trx_release_locks`. SyncProf suggests to remove the lock. However, the bottleneck is only exposed with read-only transactions; conflicts may still occur for write transactions. After manually examining the code, we suggested to check whether a transaction is read-only, and returns without acquiring the lock. We reported this optimization to the MySQL developers.

*Patterns Not Found.* For two bottlenecks, SyncProf could not match any optimization patterns. In Ocean, the real optimization is to reduce the amount of code in the CS to eliminate unnecessary lock acquire attempts [24]. The other case occurred on MySQL8, where the two logging functions (commit and write) both use the `log_sys->mutex` lock to write to a buffer. The real optimization is to use two different buffers, so the commit and write functions execute concurrently.

Both of the above cases require developer knowledge to find the optimizations. However, SyncProf is still beneficial because it identifies synchronization bottlenecks and can relieve developers from manually locating the problems before getting to specific optimizations.

#### 4.2.4 *RQ4: Efficiency of SyncProf*

The "Time" columns in Table 3 show the total analysis time of SyncProf, SyncProf without the first step that selects test cases, and DRD. Without the first step of the approach, the analysis is significantly higher because up to four times more test cases need to be executed and summarized, whereas selecting test cases accounts for less than 2% of the overall analysis time. The efficiency of DRD is similar to that of SyncProf. For some programs, DRD require less analysis time than SyncProf, primarily because it does not need additional runs for test summarization. However, this slight advantage in efficiency is outweighed by SyncProf's improvements in effectiveness (Section 4.2.2).

The runtime overhead for binary instrumentation was about 10x for open source projects, and 4x for the benchmark programs. If the search for optimizations was enabled, the overhead was up to 60x for open source projects, and 100x for the memory intensive benchmark programs. These overheads are in the same order of magnitude as that of other profilers [20,41]. We consider these overheads to be acceptable for in-house performance profiling, which is the intended usage scenario of SyncProf.

## 5. DISCUSSION

It is possible that optimizing a CS introduces a new bottleneck due to CS reordering. We controlled for this threat by verifying whether the optimization can improve overall performance. Another potential problem is the overhead of profiling. To avoid the problem that profiling influences the performance and may distort the profiling results, we do not measure wall-clock time. Instead, SyncProf uses a logical clock that counts the number of evaluated conditionals (see Section 2.2.2).

SyncProf may fail to detect a problem if the bottleneck CS is not exercised by the existing test cases. Also, SyncProf may suggest incorrect optimizations that can affect program correctness (e.g., data races due to lock elimination). In our study, the reason for the incorrect suggestions is that the test cases do not exercise all CSs needed for precise optimization. In addition, without using all inputs, the performance impact reported by SyncProf may be different from that in the deployed environment. Work on automated test input generation may address these limitations.

## 6. RELATED WORK

Several profiling techniques identify and accelerate synchronization bottlenecks using software and hardware approaches [9, 15, 17, 27, 32, 36, 56]. These techniques focus on individual executions. For example, Tallent et al. [56] use an idleness metric to locate the threads that are responsible for the idleness, so the threads can be accelerated by the hardware. Dynatrace [15] and Intel Vtune [1] report lock contention by computing thread synchronization time. Similar to DRD, they analyze individual execution, whereas SyncProf identifies summarizes the performance impact of each CS across multiple executions. None of the existing techniques tracks indirect or nested dependences of CSs, which may lead to imprecise results. Moreover, none of the above techniques suggests optimizations.

Other research finds performance problems through dynamic analysis [2, 23, 26, 41] or static pattern matching [51]. For example, MemoizeIt [57] detects repeated method calls that can be optimized through memoization. StackMine mines call stack traces to discover call sequences with a high performance impact [23]. While the above techniques are inspiring and effective, they focus on sequential programs.

Several approaches analyze concurrency-related performance issues [12,22,45,62,66,66]. For example, SpeedGun [45] generates multi-threaded performance test cases to expose performance differences between two program versions. Yu et al. [66] propose a trace-based dynamic approach to identify general performance problems (including lock contention) and report root causes. Wert et al. [62] characterize symptoms of performance problems, which can be used to determine a specific type of issue, such as synchronization-related performance problems. However, none of the above techniques localizes the root cause of a bottlenecks or suggests optimizations.

Some techniques optimize critical sections or locks to improve performance [12, 47, 48, 55, 67]. For example, Lock Elision (LE) [47, 48] uses a hardware approach to dynamically remove unnecessary locks. However, this approach does not localize bottlenecks or suggest optimizations at the code level. Curtsinger et al. [12] use a causal profiling approach to identify code with optimization opportunities. It requires developers to insert progress/delay points at the start and end of an event of interest (e.g., a transaction). Again, this approach uses single inputs and does not suggest optimizations. However, we may build upon their work to predict the benefits of optimization.

There has been some work on using lock-related graphs to achieve different goals [21, 30, 50, 61, 66]. For example, the wait-for graph and its extensions have been widely used to detect deadlocks [30, 50, 61]. Yu et al. [66] propose a wait graph to identify general performance problems in device drivers. Our SyDG approach is different in several aspects. First, the SyDG specifically targets synchronization bottlenecks and thus can effectively help developers identify ineffective synchronization usage. Second, a SyDG models several types of causal-edges, which can precisely compute the performance impact of CSs. Third, the SyDG provides a generic basis for computing multiple performance metrics.

## 7. CONCLUSIONS

We present SyncProf, a concurrency-focused performance profiler that helps developers identify synchronization bottlenecks, localize their root cause, and find suitable optimizations. The approach summarizes the performance behavior from multiple inputs and executions into a ranked list of critical sections. A key ingredient of SyncProf is a novel graph representation of the wait relations between critical sections, which provides a generic basis for metrics that summarize the performance impact of critical sections and for suggesting bottleneck-specific optimization strategies. Our study shows that the approach successfully identifies and localizes both existing and previously unknown bottlenecks, and that it suggests effective optimization strategies for most of them. Given the increasing need for efficient concurrent software, we consider our work to be a useful contribution to the developer's toolbox.

***Acknowledgments.*** We thank the anonymous reviewers for their valuable feedback. This research is supported in part by the NSF grant CCF-1464032, by the German Research Foundation within the Emmy Noether project "ConcSys" and by the German Federal Ministry of Education and Research and the Hessian Ministry of Science and the Arts within "CRISP".

## 8. REFERENCES

[1] Intel® vtune™amplifier xe, 2014. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[2] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, 2004.

[3] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *SIGMETRICS*, pages 126–137, 1996.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test. Verif. Rel.*, 13:207–227, 2003.

[5] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker. Software performance testing based on workload characterization. In *WOSP*, pages 17–24, 2002.

[6] P. Barford and M. Crovella. Critical path analysis of tcp transactions. *TON*, 9:238–248, 2001.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *PLDI*, pages 255–268, 2010.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[9] G. Chen and P. Stenstrom. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *SC*, pages 71:1–71:11, 2012.

[10] Diagnosing Lock Contention with the Concurrency Visualizer, 2010. Microsoft MSDN.

[11] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *PPOPP*, pages 15–24, 2010.

[12] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *SOSP*, pages 184–197, 2015.

[13] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *CSMR*, pages 11 pp.–70, 2006.

[14] DRD: a thread error detector, 2015. http://valgrind.org/docs/manual/drd-manual.html.

[15] Identify Thread Contention, 2015. https://community.dynatrace.com.

[16] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.

[17] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *ISCA*, pages 362–370, 2010.

[18] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[19] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, 2003.

[20] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *FSE*, pages 357–368, 2015.

[21] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, pages 428–451, 1975.

[22] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *FSE*, pages 426–438, 2015.

[23] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[24] M. Heinrich and M. Chaudhuri. Ocean warning: Avoid drowning. *SIGARCH Comput. Archit. News*, 31(3):30–32, 2003.

[25] W. Heirman, T. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *IISWC*, pages 38–49, 2011.

[26] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[27] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, pages 223–234, 2012.

[28] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, pages 19–30, 2012.

[29] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *FSE*, pages 13–22, 2009.

[30] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *SPAA*, pages 297–303, 2008.

[31] Ongoing work on lock contention in qemu driver, 2013. https://www.redhat.com/archives/libvir-list/2013-May/msg01247.html.

[32] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, pages 6–6, 2012.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[34] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[35] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[36] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. Ips-2: the second generation of a parallel program measurement system. *TPDS*, pages 206–217, 1990.

[37] D. Mosberger and T. Jin. Httperf&mdash;a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[38] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[39] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.

[40] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and

D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *ICSE*, pages 727–737, 2012.

[41] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

[42] E. Novillo and P. Lu. A case study of selected splash-2 applications and the sbt debugging tool. In *IPDPS*, pages 290.2–, 2003.

[43] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. Uts: An unbalanced tree search benchmark. In *LCPC*, pages 235–250, 2007.

[44] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *PLDI*, pages 521–530, 2012.

[45] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *ISSTA*, pages 13–25, 2014.

[46] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, 2014.

[47] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305, 2001.

[48] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *EuroSys*, pages 261–274, 2009.

[49] B. Sahelices, P. Ibáñez, V. Viñals, and J. M. Llabería. A methodology to characterize critical section bottlenecks in dsm multiprocessors. In *Euro-Par*, pages 149–161, 2009.

[50] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *PPoPP*, pages 29–42, 2014.

[51] M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *ICSE*, 2016.

[52] K. Sen. Effective random testing of concurrent programs. In *ASE*, pages 323–332, 2007.

[53] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[54] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, pages 51–64, 2011.

[55] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PLDI*, pages 11–22, 2012.

[56] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, pages 269–280, 2010.

[57] L. D. Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *OOPSLA*, 2015.

[58] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Software. Eng.*, 10(2):203–232, 2003.

[59] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[60] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPOPP*, pages 137–146, 2006.

[61] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, pages 281–294, 2008.

[62] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *ICSE*, pages 552–561, 2013.

[63] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP*, pages 602–629, 2005.

[64] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[65] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14, 2005.

[66] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206, 2014.

[67] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin. On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach. In *CGO*, 2015.