Empir Software Eng
https://doi.org/10.1007/s10664-017-9578-1

Pinpointing and repairing performance bottlenecks in concurrent programs

Tingting Yu[1] · **Michael Pradel**[2]

Abstract Developing concurrent software that is both correct and efficient is challenging. Past research has proposed various techniques that support developers in finding, understanding, and repairing concurrency-related correctness problems, such as missing or incorrect synchronization. In contrast, existing work provides little support for dealing with concurrency-related performance problems, such as unnecessary or inefficient synchronization. This paper presents SyncProf, a profiling approach that helps in identifying, localizing, and repairing performance bottlenecks in concurrent programs. The approach consists of a sequence of dynamic analyses that reason about relevant code locations with increasing precision while narrowing down performance problems and gathering data for avoiding them. A key novelty is a graph-based representation of relations between critical sections, which is the basis for computing the performance impact of a critical section and for identifying the root cause of a bottleneck. Once a bottleneck is identified, SyncProf searches for a suitable optimization strategy to avoid the problem, increasing the level of automation when repairing performance bottlenecks over a traditional, manual approach. We evaluate SyncProf on 25 versions of eleven 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 repair the bottlenecks.

Keywords Testing · Concurrency · Performance bottlenecks

Communicated by: Martin Monperrus and Westley Weimer

✉ Tingting Yu
tyu@cs.uky.edu

Michael Pradel
michael@binaervarianz.de

[1] Department of Computer Science, University of Kentucky, Lexington, KY, USA

[2] Department of Computer Science, TU Darmstadt, Darmstadt, Germany

Published online: 27 November 2017

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 (Gu et al. 2015). Another study shows that unnecessary synchronization is a common root cause for real-world performance problems (Jin et al. 2012).

Over the past decades, most research has focused on analyzing and debugging the correctness of concurrent programs, e.g., through detecting data races (Kahlon et al. 2009; Marino et al. 2009; Bond et al. 2010; Effinger-Dean et al. 2012) or atomicity violations (Artho et al. 2003; Xu et al. 2005; Wang and Stoller 2006; Flanagan and Freund 2004; Shacham et al. 2011), schedule exploration (Visser et al. 2003; Sen 2007; Musuvathi et al. 2008; Sen 2008; Coons et al. 2010; Burckhardt et al. 2010), test generation (Pradel and Gross 2012; Nistor et al. 2012; Choudhary et al. 2017), and static analysis (von Praun and Gross 2003; Flanagan and Qadeer 2003; Williams et al. 2005; Naik et al. 2009; Joshi et al. 2012). In contrast, the problem of detecting and repairing concurrency-related performance problems is currently understudied.

For example, consider a performance problem in the `libvirt` KVM/QEMU driver (Ongoing work on lock contention in QEMU driver 2013). A user reports a slowdown in virtual machine creation when multiple virtual machines are created in parallel. Figure 1 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. In the example, thread T1 obtains lock L1 and enters CS1. Next, T2, T3, and T5 all attempt to obtain L1 and thus are serialized. Before attempting to obtain L1, T2 obtains L2 and enters CS2. The execution of CS3 is nested inside CS2, i.e., T2 acquires L1 while holding L2. Once CS2 is finished, T4 acquires L2 and enters CS5.

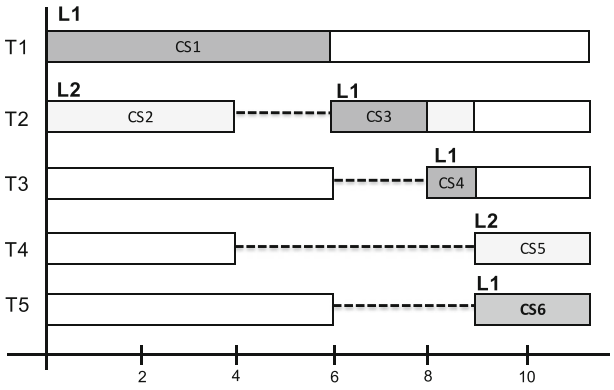


Fig. 1 A motiving example

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 fixed the problem by splitting the lock L1 into two locks. We will use the threads and CSs in this example to illustrate our approach in Section 3.

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 repair 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 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 (Diagnosing Lock Contention with the Concurrency Visualizer 2010) 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 Fig. 1, 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 that 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 article 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. Next, SyncProf reports a ranked list of likely root causes of synchronization bottlenecks, along with their descriptions (e.g., code locations). Finally, 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 developers 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. Instead, it leaves the final decision if and how exactly to address a synchronization bottleneck to the developer. Despite this limitation, the approach significantly improves the level of automation of repairing synchronization bottlenecks compared to the traditional, purely manual approach.

We envision that the approach can be used in at least three scenarios. First, a developer that is not aware of any synchronization bottlenecks in a program may profile the program with SyncProf to check if any such a bottleneck exists. Second, a developer that knows that a program suffers from a concurrency-related bottleneck can use SyncProf to localize the problem. Third, a developer looking for a way to avoid a bottleneck can use SyncProf to obtain a concrete suggestion of an optimization strategy tailored to the specific bottleneck. These usage scenarios are similar to traditional testing, fault localization, and program repair, 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 speedups between 1.17 and 2.6.

In summary, this article contributes the following:

- A concurrency-focused profiler that detects synchronization bottlenecks, pinpoints their root causes, 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.

This article extends and refines a previously presented conference paper (Yu and Pradel 2016). Specifically, we introduce barrier synchronization and extend our algorithm to handle this new synchronization type. We also propose a new bottleneck optimization strategy that can reduce the size of a critical section to eliminate unnecessary bottlenecks. Furthermore, we perform a more thorough empirical study on a more substantial base of artifacts. Finally, we made a large number of changes related to the presentation of the article and included more detailed descriptions of the algorithm, evaluation, and related work.

In the next section we present a motivating example and background. We then describe SyncProf in Section 3. Our evaluation follows in Sections 4 and 5, followed by a discussion in Section 6. We present related work in Section 7, and end with conclusions in Section 8.

2 Motivation and background

2.1 A motivating example

The example discussed illustrated in Fig. 1 was well tested regarding functional correctness. However, it takes a long time for OpenStack to create virtual machines (VMs) when many VMs are requested in parallel. In the process of replicating this bug, we found that when many VM creation threads (e.g., 20 threads) are spawned, the system is significantly slowed down and the CPU usage is low (less than 60%). Specifically, when a domain is created, T1 obtains the domain object's lock `DomObj` and enters critical section CS1. At the same time, T2 launches a server thread and obtains the security lock `VirSecMan`. Next, T2, T4, and T5 all attempt to obtain `DomObj` and thus are serialized. The fix is to decrease the granularity of the `DomObj` lock to make it more fine grained, i.e., to split the critical section protected by `DomObj` into multiple critical sections protected by different locks. This change will avoid `domainCreateFlags` (i.e., options for creating VM domains) to be unnecessarily serialized with other critical sections holding the same lock.

While the execution scenario of this example is straightforward, the developers had a difficult time in locating the root cause of the bottleneck due to several reasons. First, existing techniques that measure thread idleness or critical section execution time fail to track dependencies between the critical sections. For example, a profiler based on idleness measurement reported that the function `ThreadPoolWorker` (CS5) on T4 was problematic. However, CS5 is not the root cause, as optimizing it did not improve performance. A profiler that measures the time spent inside each critical section reported that the function `NetServerDispatch` (CS2) on T2 was the most critical (i.e., 90 time units). Again, optimizing CS2 did not improve the performance much either, as CS3 still has to wait for CS1 to finish. The true root cause of the synchronization bottleneck turns out to be the function `domainCreateFlags` (CS1) on T1. Second, multiple profiles were generated given different inputs, workloads, and system configurations. As such, it took substantial manual effort to analyze a number of traces to locate the bottlenecks optimization. Third, once the root cause was located, developers spent additional time figuring out how to fix the problem.

2.2 Synchronization bottlenecks

We define a *synchronization bottleneck* as an execution point where threads contend to reach a synchronization point, causing a significant impact on the overall execution time. A synchronization bottleneck may consist of a single thread or multiple threads that need to reach a synchronization point before other threads can make progress. Threads that are waiting for a synchronization point are called *waiters*, and threads that are executing a synchronization point are called *executers*. A single instance of a synchronization bottleneck may be responsible for multiple waiters. The consequence of a synchronization bottleneck can be substantial because every cycle a synchronization bottleneck is running causes its waiters to waste one cycle. Synchronization bottlenecks may involve several kinds of synchronization operations. We consider operations that are widely used and likely to incur contention (Joao et al. 2012), including: (1) mutexes (`pthread_mutex_{lock, unlock}`), (2) binary semaphores (`sem_{wait, post}`), (3) spin locks (`pthread_spin_{lock, unlock}`), and (4) barriers (`pthread_barrier_wait`). The synchronization operations can be classified into two categories.

Lock-based synchronization The first three synchronization operations (1), (2) and (3) fall into this category. They are similar in the sense that they ensure mutual exclusion through CSs. A CS is a code segment that must be executed by one thread at a time, and any other threads wanting to execute it must wait. A CS can become a synchronization bottleneck if it contends with CSs in other threads. Therefore, in the case of lock-based synchronization, there is only one executer and possibly multiple waiters for a synchronization bottleneck. A lock operation involves three steps — acquire the lock, obtain the lock, and release the lock, so a CS is enclosed by the instruction to obtain the lock and the instruction to release the lock. The time spent between acquiring and obtaining a lock is the wait time for the CS protected by the lock object. Figure 1 shows the execution of five threads running six critical sections with two lock-based synchronization objects.

Barrier-based synchronization For barrier-based synchronization, i.e., category (4) above, all previously arrived threads will be blocked until the last thread reaches the barrier. The *barrier region* is defined as the code segment along each path preceding the barrier (Gupta and Epstein 1990). In this case, a critical section refers to a barrier region itself or the code segment succeeding the barrier region. The time difference between barrier entry (e.g., when `pthread_barrier_wait` is executed) and barrier exit (e.g., when `pthread_barrier_wait` returns) is the wait time for the thread on that barrier. Figure 2 shows the execution of T1–T4 on a barrier. T1 and T4 reach the barrier at time 6 and start waiting on T2 and T3. T2 reaches the barrier at time 10, leaving T3 as the only running thread until time 12, when T3 reaches the barrier and all four threads can continue. The example involves eight CSs. CS3 takes the most time to execute (i.e., 12 time units), whereas CS1 and CS4 introduce the most wait time (i.e., 6 time units). There are two synchronization bottlenecks (CS2 and CS3), where each synchronization bottleneck contends with other threads to reach the barrier. Specifically, CS2 contends with CS5 and CS8, and CS4 contends with CS5, CS6, and CS8. Therefore, in the case of barrier synchronization, there can be multiple executers and multiple waiters for a synchronization bottleneck.

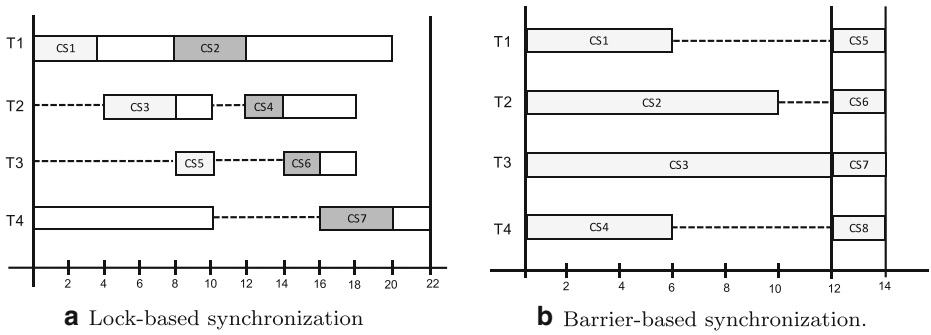


Fig. 2 Examples of synchronization bottlenecks

3 Approach

This section presents SyncProf, a profiling approach that helps developers identify, localize, and optimize synchronization bottlenecks. Figure 3 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 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

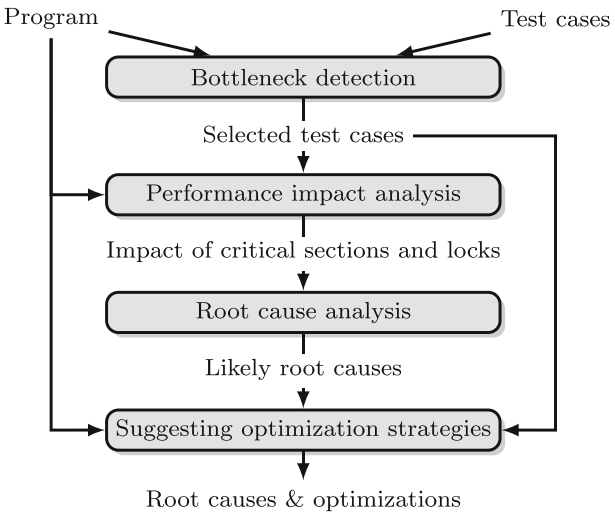


Fig. 3 Overview of the SyncProf approach

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.

The second and the last part are based on a graph-based representation of the relations between CSs, which SyncProf extracts from program executions. The graphs provide a generic representation that enables SyncProf to compute metrics that summarize the performance impact of CSs, and to match related CSs against known bottleneck patterns. To deal with the non-deterministic performance behavior that is inherent to concurrent programs, SyncProf repeatedly executes tests and performs statistical analyses to identify the "typical" performance impact of each CS and lock.

3.1 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 execution (Mosberger and Jin 1998; Draheim et al. 2006; Arlitt and Williamson 1996); 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 (Avritzer et al. 2002).

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 (Yu et al. 2014), such as CPU intensive operations (e.g., loops). Thus, additional symptoms are needed to isolate traces that contain synchronization bottlenecks.

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 $\mathcal{M}'$ is 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 δ_{max} rounds (default: $\delta_{max} = 50$), the approach discards the test case. We discuss the effectiveness of selecting test cases when using different parameters in Section 6.

An alternative to our approach would be to identify bottlenecks by measuring the percentage of time spent on synchronization. However, this alternative approach can impose a significant runtime overhead because it requires instrumenting the program. Instead, our approach is lightweight, enabling the first part of SyncProf to consider many tests and workload sizes.

3.2 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 between 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.

3.2.1 Synchronization dependence graph

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

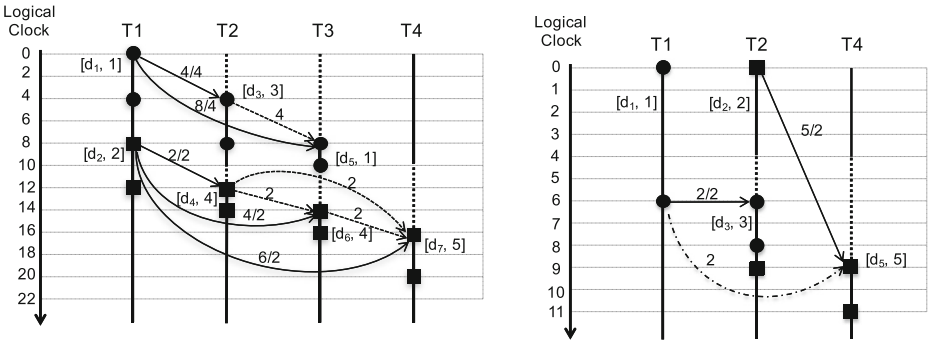
Definition 1 A synchronization dependence graph (SyDG) is a graph (V, E). V is a set of nodes $\{d_1, d_2, \ldots d_i\}$, where d_i is a dynamic instance of a CS. E is a set of directed causality edges $E = \{d_i \rightarrow d_j\}$, implying d_j is waiting for d_i.

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 in thread T_i, which originally waits for d_j in thread T_j, now waits for a new dynamic CS instance d_k in thread T_k. For instance, in Fig. 2a, CS6 indirectly waits for CS4 because it originally waits for CS2. 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 4a displays a SyDG with two subgraphs from the example of Fig. 2a. 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 4b displays a partial SyDG for Fig. 1 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).



a Synchronization dependence graph for Figure 2a.

b Nested critical section for (a).

Fig. 4 Examples of partial synchronization dependence graphs for Fig. 1

3.2.2 Constructing the graph

SyncProf constructs the SyDG for an execution by instrumenting the program and by analyzing the current execution trace. An execution trace is a sequence of synchronization events, i.e., acquiring a lock, obtaining a lock, releasing a lock, and a barrier. Figure 5

```
procedure BuildSyDG (trace)
1: V = φ ; E = φ
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) ← d.t
9:     case lock obtain:
10:        setActive (d.tid) ← true
11:        for each d' ∈ in(d)
12:          c(d', d) ← d.t - c(d', d)
13:          for each d'' ∈ (out(d')/(d', d))
14:            E.addEdge(d, d'')
15:            c(d, d'') ← d.t
16:            c(d', d'') ← d.t - c(d', d'')
17:          for each o ∈ active(d.tid)/d
18:            for each d'' ∈ out(o)
19:              E.addEdge(d', d'')
20:              c(d',d'') ← c(d', d)
21:     case lock release:
22:          setActive (d.tid) ← false
23:     case barrier:
24:          V.addNode(d)
25:          for each d' ∈ V/d
26:            if d.bar = d'.bar
27:              E.addEdge (d', d)
28:              c(d',d) = d.st - d'.ct
```

Fig. 5 Algorithm to compute the SyDG

describes the algorithm for constructing the SyDG on the fly. Whenever a synchronization event is executed, the algorithm updates the SyDG nodes, direct/indirect waiting edges and their corresponding costs.

Adding a node When a `lock acquire` event (attempting to enter a CS) 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). At this point, the current thread does not enter the CS yet, so the status of the CS is inactive by default. Specifically, a CS becomes active on the current thread upon obtaining a lock (line 10) and inactive upon releasing a lock (line 22). In the case of barrier synchronization, a node is created when a barrier is reached (line 24).

Adding a direct waiting edge 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 that d' spends on waiting for d. The cost $c(d', d)$ is temporarily set to the current time (line 8), which will be used to calculate the actual cost of $c(d', d)$ when d' obtains the lock. When d obtains the lock, the algorithm iterates through the in-going edges of d and finds the CSs that d was previously waiting for, i.e., waiters of d. The cost of each edge is updated by the time difference between the current time and the time when d started waiting (line 12). In the example of Fig. 4a, when d_5 is entered, the cost of edge (d_1, d_5) is 8 (shown on the left of the slash).

In the case of barrier synchronization, a direct waiting edge is added from each wait event d' to the current event d, indicating that the two events are waiting on the same barrier (line 27). The cost of each edge is the time elapsed since d started waiting for d'. Consider the example of Fig. 2, CS5 is blocked at time 6 and waits for the other threads to reach the barriers. When CS2 reaches the barrier, the cost of (CS5, CS2) is 4 because CS5 spends 4 time units waiting for CS2. When CS3 reaches the barrier, the cost of (CS5, CS3) is $12 - 6 = 6$.

Adding an indirect waiting edge The algorithm computes the indirect edge by iterating through each CS d'' that was previously waiting for 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, the direct waiting edge (d', d'') is updated by subtracting the old edge cost from the current time (line 16) because d'' is currently waiting on d. In the example of Fig. 4a, 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 the direct waiting edge (d_1, d_5) is then reduced from 8 to 4 (shown on the right of the slash).

Note that the barrier synchronization does not involve indirect edges because all CSs converge at the same barrier point. In other words, a CS can directly wait for multiple CSs at the same time. For example, in Fig. 2, CS5 directly waits for CS2 and CS3.

Adding a nested waiting edge To construct a SyDG 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 waiting on d' (on which d is waiting). Thus, the algorithm adds a nested waiting edge from d' to each waiter d'' of o, indicating that d'' is currently waiting for d'. The edge cost of (d', d'') is updated to (d', d) because d'' spends the same waiting time as d on waiting for d'. For example, in Fig. 4b, 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 current implementation of SyncProf does not support the nested waiting edge for barrier synchronizations. While it is possible that barriers can be nested, we do not find such cases in our studied subjects.

The timestamps used to construct the SyDGs are based on a logical clock, which measures the number of executed conditionals, including direct/indirect calls and direct/indirect branches, instead of actual wall-clock execution time. A direct/indirect call is considered to be a conditional because it involves runtime cost, so we account for the cost by increasing the conditional counter. The rational for measuring time through this proxy metric, which is inspired by Pradel et al. (2014), 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 the 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.

3.2.3 Performance impact metrics

Based on the SyDG of an execution, SyncProf quantifies the performance impact of the 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 (Joao et al. 2012; Chen and Stenstrom 2012; Tallent et al. 2010) 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 Fig. 4a. 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, but the results may be misleading when the synchronization bottleneck does not impact the completion time of a program. To address this problem, the CPWT metric considers the critical path[1] (Barford and Crovella 2001; Chen and Stenstrom

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

2012) 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 2 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 Fig. 4a, 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 indicates that CS2 induces the highest performance impact because it is in the critical path.

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, performance bottleneck optimization is done for multiple CSs associated with the same lock (Zheng et al. 2015). As such, 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 Fig. 4a, where the two SyDGs involve different locks $L1$ (top subgraph) and $L2$ (bottom subgraph). Thus, $PI_{T,L1} = 4 + 4 + 4 = 12$, and $PI_{T,L2} = 2 + 2 + 2 + 2 + 2 + 2 = 12$.

3.2.4 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 3.2.3). At the end of each repetition, SyncProf accumulates $PI_{T,CS}$ using a set $M_{T,CS}$, i.e., $M_{T,CS} = \{PI_{T,CS_1}, PI_{T,CS_2}, \ldots, PI_{T,CS_n}\}$, where n is the n^{th} repetition and $n \leq N$. Next, the standard deviation of $M_{T,CS}$ is calculated, denoted by $\sigma(PI_{T,CS})$. The standard deviation is expected to decrease as more data points for $PI_{T,CS}$ are collected. SyncProf stops repeating executions when $\sigma(PI_{T,CS})$ is below a specified threshold δ_{stop} or when n reaches the maximum value N. By default, SyncProf sets δ_{stop} to the percentage (σ_p) of the mean of the M_n, that is, $\overline{M_{T,CS}} \cdot \sigma_p$ (Pradel et al. 2014). 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 δ_{stop} after the number of repetitions reach N. In this case, SyncProf will report the CS CS as inconclusive (Pradel et al. 2014) and leave it for manual inspection. However, we did not find any inconclusive test cases in our evaluation. The final $\text{PI}_{T,CS}$ is the mean of $M_{T,CS}$, excluding any inconclusive test cases, i.e., $\text{PI}_{T,CS} = \overline{M_{T,CS}}$.

Another source of nondeterminism involves cache misses, which can greatly increase response times of SyncProf (Lozi et al. 2012). This can happen when a CS accesses a shared resource that has recently been accessed by another core. To address this issue, before the

Table 1 Example of a 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	...	...	...	...

training process starts on each test j, SyncProf repeatedly executes j for W times to warm up the cache.

3.3 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 $PI_{CS} = \{\overline{M_{1,CS}}, \overline{M_{2,CS}}, \ldots, \overline{M_{T,CS}}\}$ 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 an ANOVA test[2] (Mertler and Vannatta 2002) among all CS sets (e.g., PI_{CS}s) and to conclude that one CS has more performance impact than another when there is a statistically significant difference. We rejected this idea because this approach ranks CSs in a partial order, which could be difficult for developers to understand.

3.4 Suggesting optimization strategies

The final step of SyncProf heuristically suggests optimization strategies for the detected synchronization bottlenecks. We consider four optimizations: *unnecessary synchronization elimination*, *lock split*, *reader-writer locks*, and *CS reduction*. 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 similarly to the SyDG (Section 3.2), except for three differences. First, a SyDG$_o$ does not record or update timestamps because timing information is not needed when optimizing the CSs that deal with positioning locks. Second, 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. Third, besides instrumenting CS entry and exit points and synchronization operations when generating a SyDG, SyncProf instruments memory reads and writes for constructing a SyDG$_o$. To mitigate runtime overhead, SyncProf instruments only the top K CSs in the ranking and the

[2] ANOVA tests the significance of group differences between two or more groups

CSs that use the same lock as these K CSs; this information can be retrieved by analyzing the original SyDGs. The 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 the set of all SyDG$_o$s into a *universal synchronization dependence graph* ($USyDG$). As in the SyDG, a node in the USyDG is a dynamic instance of a CS and an edge indicates the a wait relation between CSs. However, a USyDG describes the wait relations between CSs across *all threads over all test cases*. As a result, the optimization can be performed on a single graph. One challenge in building the USyDG involves merging dynamic CS instances across multiple executions because thread IDs may vary across different executions. To address this challenge, SyncProf uses the static thread ID as the key, and the dynamic CS instances are merged only when they share the same static ID on the same thread.

Figure 6 shows the algorithm for computing the USyDG. The algorithm first constructs a matrix M for program P where each row indicates a SyDG$_o$ and each column indicates a dynamic thread ID. A dynamic thread ID is denoted as $m.n$, where m is the static thread ID and n is the index of the dynamic instance of m. The initial value of n is 0, and n is increased when a new instance of m is created. An element in the matrix M indicates a list of CSs on the thread $m.n$ of a SyDG$_o$s.

To construct a $USyDG$, for each thread across all SyDG$_o$s (i.e., each column of M), 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 7a and b display two SyDG$_o$s from two tests t_1 and t_2. Table 2 shows the corresponding SyDG$_o$ matrix. The first row indicates the threads (i.e., $m.n$). Each cell in rows 2–3 indicates the static CS IDs on a thread of a SyDG$_o$. There are six static CSs with a common lock. Next the CSs on the same thread for all SyDG$_o$ are merged. Figure 7c displays the USyDG by merging $SyDG_{o_1}$ and $SyDG_{o_2}$. The static CSs with common identifiers SyDG$_o$ are merged into one node (i.e., node 1 in T1 and node 5 in T4) and their read and write sets are joined. 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_{wt} = {w}). For example, because node 1 in SyDG$_{o_1}$ and node 1 in SyDG$_{o_2}$ involve the same shared variable read, node 1 in USyDG contains a read set CS_{rd} = {r}. The same with node 5 and other nodes. (L) indicates a CS is protected by a lock L.

3.4.1 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*, (3) *low-degree-conflicts*, and (4) *long-critical-section*.

procedure *BuildUSyDG* ($\{SysDG_o\}$)
1: V = ϕ ; E = ϕ
3: construct $M[S][T]$
4: **for each** t **in** T
5: **for each** s **in** S
6: USyDG[t] $\leftarrow$ USyDG[t] $\cup$ $M[s][t]$
7: add edges

Fig. 6 Algorithm to compute the USyDG

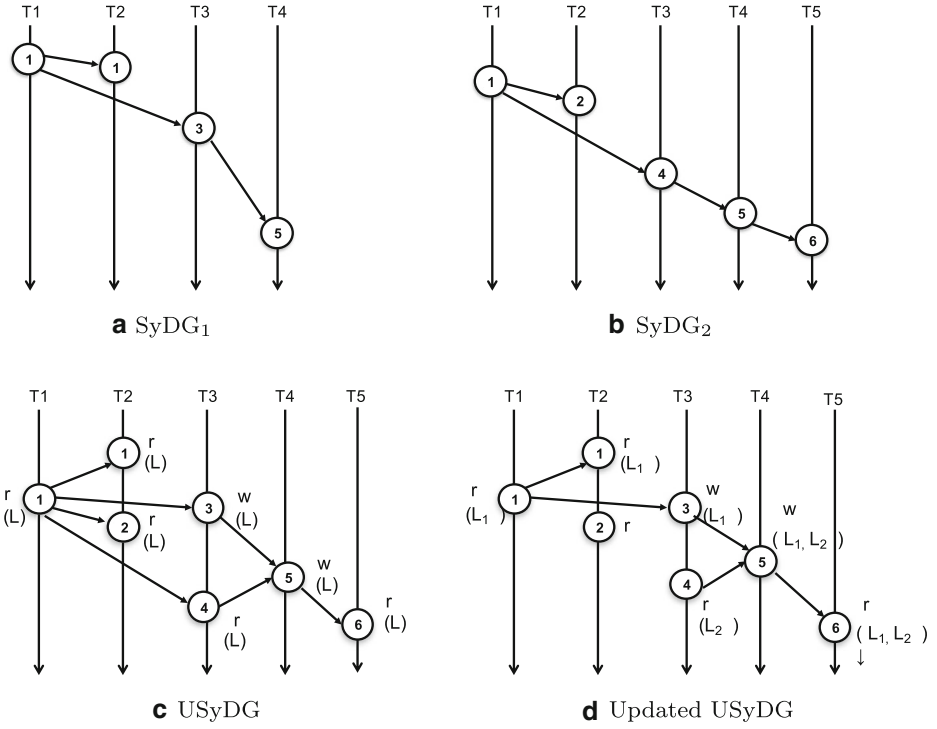


Fig. 7 Optimization example

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 $(CS_{rd_m} \cap CS_{wt_n} = \phi) \wedge (CS_{rd_m} \cap CS_{rd_n} \neq \phi)$.

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 location, 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 δ_{deg}, the CS pair matches the *low-degree-write* pattern. We set $\delta_{deg} = 80\%$ as a default.

The *long-critical-section* pattern happens when a CS pair matches none of the patterns (1), (2) and (3), but a number of consecutive accesses in a CS do not involve conflict accesses. For example, a CS contains 100 instructions, but only the last 20 instructions involve conflict accesses. In this case, the first 80 instructions do not need to be protected

Table 2 Example of SyDG_o matrix

	1.1	1.2	1.3	1.4	1.5
SyDG_{o_1}	1	1	3	5	–
SyDG_{o_2}	1	2	4	5	6

by the CS. To identify this pattern, SyncProf begins with the CS entry and tracks the number of consecutive memory accesses that do not contain conflicts, denoted by M_e. SyncProf then analyzes the trace that records memory accesses in a backward way from the CS exit to the last conflict access in the CS; this number is denoted by M_x. If the ration between the sum of M_e and M_x and all memory accesses exceeds a threshold δ_{long}, the CS pair matches the *long-critical-section* pattern. We set $\delta_{long} = 20\%$ as a default.

3.4.2 Suggesting concrete optimizations

SyncProf can suggest four types of concrete optimizations for the detected optimization patterns.

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. If the static CS identifiers are identical, the edge is removed only when the identifiers do not connect with other nodes that have different CS identifiers. In the example of Fig. 7c, edges (T1:1,T2:2), and (T1:1,T3:4) are removed (updated in Fig. 7d), because the nodes in both edges match the *read-read* pattern. The edge (T1:1,T2:1) is not removed because the 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 Fig. 7d, SyncProf suggests to remove node 2 in T2.

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 Fig. 7d, 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 to use reader-writer locks instead of locks that enforce full mutual exclusiveness. 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 identifier of the last node of the path is a reader of s. In Fig. 7d, <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.

Critical section reduction For each simple dynamic path in the CS, SyncProf identifies the first and last conflict accesses in the CS, denoted by SV_e and SV_x, respectively. SyncProf groups the paths that have conflict accesses and their SV_es belong to the same basic block. For each group, SyncProf picks the shared variable (denoted by SV_e') that is closest to the CS entry and moves this entry to the location right above the statement involving SV_e'. In

Fig. 8, there are two simple paths that contain conflict accesses: p_1 = <1-4, 6-7> and p_2 = <1-5, 6-7>. Since the conflict access at node 3 is closest to the CS entry (node 1), the `lock` at node 1 is moved to the position above the node 3. Next, SyncProf filters out the simple dynamic paths that traverse the newly assigned lock. SyncProf groups the paths in which their SV_xs belong to the same basic block. For each group, SyncProf picks the shared variable (denoted by SV_x') that is closest to the CS exit and moves this exit to the location above the statement involving SV_x'. In Fig. 8, the SV_xs on the two simple paths do not belong to the same basic block. Specifically, the SV_x on p_1 is node 3 and that on p_2 is node 5. Thus, the `unlock` is assigned to each path separately. The algorithm guarantees that the CS entry and exit points are properly paired after optimization.

Handling the low-degree-write pattern In practice, it is almost impossible to suggest concrete optimizations for the *low-degree-write* pattern, as it often depends on the specific program and thus requires developers' knowledge to optimize the code. For example, developers may choose to set flags to enable synchronizations under certain condition, applying specific data structures (e.g., non-blocking algorithm Micheal and Scott 1996). In this case, SyncProf only reports CSs that match pattern (3), and leaves them for developers to investigate.

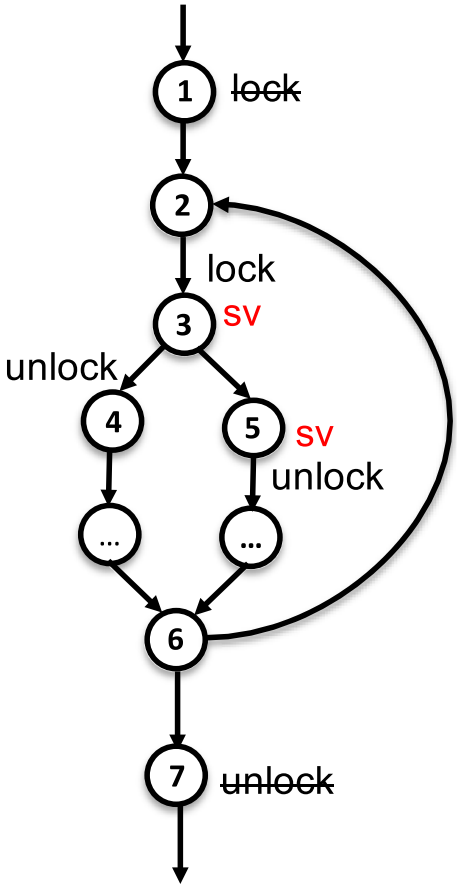


Fig. 8 An example for illustrating the critical section reduction

Note that the four types of optimizations can also be applied to barrier synchronizations because critical sections involving barriers can be encoded into the *USyDG*. For example, in Fig. 2, if CS5 and CS2 – CS4 match the pattern (1) (i.e., *null-shared*), SyncProf would suggest to remove the barrier in CS5 because it will not conflict with the other CSs. Nevertheless, the methods of eliminating, splitting, and reducing barrier regions are different from locks, which are left for developers to investigate.

3.5 Implementation

SyncProf is implemented on top of the PIN instrumentation framework (Luk et al. 2005). To obtain CS identifiers, we use CODESURFER[3] to perform context-sensitive and flow-sensitive analysis that enumerates the paths enclosed in each CS. SyncProf monitors system executions and generates information that can be recorded for use in either online or offline analysis. 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 Empirical study

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-the-art profiler?
RQ3: How effective is SyncProf at suggesting profitable optimizations?
RQ4: How efficient is SyncProf?

The first research question allows us to evaluate the effectiveness of the root cause analysis in SyncProf. The second research question evaluates SyncProf by comparing it to a state-of-the-art lock contention profiling tool. The third research question lets us further investigate whether SyncProf can effectively suggest optimization strategies. The fourth research question explores the cost of SyncProf in the root cause analysis and optimization suggestion.

4.1 Experimental setup

4.1.1 Subject programs

Table 3 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 (indicated by $*$). Table 3 also lists the numbers of lines of non-comment code in the applications (Column 2) and the sources of bottlenecks (Column 3).

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

Table 3 Subject programs

Program	NLOC	Issue	T	T_{sel}	CS	CS_{cov}	OBJ	OBJ_{cov}
UTS	4.9K	(Chen and Stenstrom 2012)	16	8	17	17 [8, 14]	6	6 [6, 6]
Radiosity	8.2K	(Chen and Stenstrom 2012)	22	14	54	54 [21, 32]	6	6 [6, 6]
Ocean	2.6K	(Heinrich and Chaudhuri 2003)	44	21	44 (40)	44 [25, 30]	15	15 [13,15]
Barnes	2.0K	(Woo et al. 1995)	12	9	12 (6)	12 [9, 12]	4	4 [4, 4]
Cholesky∗	3.7K	–	21	12	10 (4)	10 [5,6]	4	4 [4, 4]
FMM∗	3.2K	–	26	0	41 (13)	41 [21, 24]	9	9 [9, 9]
Raytrace	6.1K	–	22	12	18 (3)	18 [13, 15]	6	6 [6 ,6]
Water	1.2K	–	34	19	17 (9)	17 [14, 17]	6	6 [6 ,6]
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]
Bitcoin1	243.3K	2840	91	61	205	92 [21, 26]	26	16 [8, 12]
Bitcoin2	273.2K	7112	88	32	306	121 [25, 33]	32	22 [12, 19]
Bitcoin3	332.8K	4795	95	40	235	95 [22, 29]	26	16 [10, 13]
Bitcoin4	259.4K	6688	84	39	291	109 [24, 30]	29	19 [10, 15]

NLOC=#lines of code. Issue=known synchronization bottleneck from existing literatures and bug IDs, 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

The first eight programs are benchmark programs used by others to study concurrency and performance (Heirman et al. 2011; Novillo and Lu 2003; Olivier et al. 2007; Sahelices et al. 2009; Chen and Stenstrom 2012; Woo et al. 1995). UTS performs an exhaustive search on an unbalanced tree (Olivier et al. 2007). The other seven benchmark programs are from the SPLASH-2 benchmark suite (Woo et al. 1995). Only the first four benchmark programs have previously known bottlenecks (Sahelices et al. 2009; Chen and Stenstrom 2012; Woo et al. 1995). The remaining programs (i.e., Firefox, MySQL, and Bitcoin) are different versions of popular open source projects, each with known synchronization bottlenecks.

The subject programs cover various application spectrums — one of the world's most widely used web browsers, the world's most popular database engine, and a widely used payment system. To identify these bottlenecks, we search the project's issue trackers for key words "mutex/lock/synchronization contention" and "performance", manually filter the issue descriptions for synchronization-related performance problems. We then randomly selected ten MySQL issues from the latest two major versions (i.e., 5.6.x and 5.7.x), three Firefox issues, and four Bitcoin 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 $*$.

4.1.2 Test cases

For each program, we gather several test cases and workload sizes. For the benchmark programs, the workload can be specified as small, medium, or large. For example, the test case "`./Radiosity -p 2 -batch -room`" uses batch mode (does not display the rendered image) and 2 threads to compute the distribution of light in a scene. Here, the regular configuration option is `-batch` the workload attributes are the number of threads (`-p 2`) and the input size (`-room`). The thread number is increased by 1 and input size can be increased to `-largeroom`.

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.[4] For example, a mysqlslap test case "`mysqlslap --user=sysadmin --password --host=localhost --concurrency=50 --iterations=10 --auto-generate-sql-write-number= 1000`" uses 50 concurrent connections, and has 1000 row inserts run 10 times. The options `--concurrency` and `--auto-generate-sql-write-number` are workloads, where the former is always increased by 10 and the latter is increased by 100. The other options are regular inputs. For Firefox, we write Selenium tests that open multiple tabs with popular web sites simultaneously. The workload size is the number of requested sites, and is increased by 10. For Bitcoin, which accepts configuration files, we used 2-way covering arrays (Fouché et al. 2007) to generate test cases. For example, a configuration that sets "`maxconnections = 128`, `server=1`, `rpcthreads=8`, `keypool=300`, and `rpckeepalive=false`" indicates a test case. It uses the command `bitcoin-cli` to send RPC request to `bitcoind`. As such, the number of requests is the workload size (e.g., `./bitcoin-cli --regtest gettransaction ccfb9f516f`), which is increased by 10.

Table 3 gives the number of test cases for each program (column 4), the number of test cases selected as bottleneck-exposing test cases (column 5), and the number of static CSs (column 6) and static synchronization objects (column 8), as well as how many of these are covered by the test cases (columns 7 and 9). The number in each parenthesis of column 6 indicates the number of barrier regions. Because the number of covered CSs and synchronization objects may differ between different executions, we give the mean and the confidence interval. We set the confidence level (i.e., p-value) as $\alpha = 0.05$.

[4] http://www.jonahharris.com/osdb/mysql/mysql-performance-whitepaper.pdf.

4.1.3 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.

4.1.4 Evaluating the effectiveness in identifying bottlenecks

To answer RQ1 (i.e., the effectiveness of SyncProf in identifying synchronization bottlenecks), we measure the rank number (position) of CSs and locks that contain synchronization bottlenecks. We also use the rank number to compute the percentage of critical sections or locks that need to be examined to find a synchronization bottleneck in the program. This ranking strategy has been widely adopted by existing fault localization techniques (Jones and Harrold 2005; Cleve and Zeller 2005).

4.1.5 Comparison with Valgrind lock contention profiler

To answer RQ2, we compare SyncProf to a state of the art profiler to detect synchronization-related bottlenecks, the Valgrind lock contention tool DRD (2015). The DRD tool computes mutex contention by measuring the amount of time that a CS is blocked trying to acquire a lock. We choose DRD as a baseline because it is open source and extensible. DRD profiles individual test executions, i.e., applying it to the test cases from Table 3 results in many execution profiles. We apply DRD to all test cases and selected test cases, respectively.

4.1.6 Evaluating effectiveness of SyncProf in suggesting optimizations

To answer RQ3 (i.e., the effectiveness in suggesting beneficial optimizations), we compare the optimization strategy suggested by

SyncProf to the optimizations known from previous work and from the developers' fixes for the reported issues. To determine whether a SyncProf' optimization strategy matches the known optimization (i.e., ground truth), we manually examined the solution discussed in the corresponding issue report and the patch used for fixing the issue. We also examined the code locations of CSs in the patch to confirm that they match the optimization descriptions suggested by SyncProf.

4.1.7 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 subjects 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. Finally, SyncProf works on programs using pthread libraries. The general idea of our approach

could translate to programs using other concurrency constructs and methods (e.g., optimistic concurrency control).

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.

5 Results and analysis

5.1 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 4 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 three previously unknown and valid bottlenecks in Cholesky, Raytrace, and Water. Furthermore, SyncProf found another two bottlenecks that were previously unknown to us (MySQL4-2 and MySQL9). MySQL4-2 has been fixed independently of us in the next version of MySQL. We reported the bottleneck in MySQL9 to the MySQL developers (bug #80101) and this bug has been linked to an existing bug #76728.

For all 24 detected bottlenecks, SyncProf guides the programmer to the desired CS by examining at most 5% of all CSs. In fact, for nine of 24 bottlenecks using the CPWT metric, the root cause is ranked *first* among all CSs. For six out of 24 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 21 of 24 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.

We further examined the three bottlenecks (i.e., MySQL4-2, MySQL8, and MySQL9) that were not ranked at the top using the APLT metric. While the top ranked CSs have the highest performance impact values, they do not cause low CPU usage (i.e., the second and third conditions for revealing bottlenecks described in Section 3.1 are not met). In fact, the higher ranked CSs in the three programs involve SQL queries and caching in the presence of large number of queries, which are normal time-consuming operations. These cases are further validated in our bottleneck optimization step; the symptom of the synchronization bottleneck disappeared after we optimize the bottlenecks identified by SyncProf.

Table 4 SyncProf's effectiveness and efficiency in localizing bottlenecks

Bottleneck	SyncProf (%)				Time	SyncProf w/o selection			Time
	CPWT	APWT	APLT	AVG	(min)	CPWT	APWT	APLT	(min)
UTS	1 (0)	1 (0)	1 (0)	1 (0)	49.5	2 (0.9%)	2 (0.9%)	2 (0.9%)	86.1
Radio.	1 (0)	1 (0)	1 (0)	1 (0)	38.6	1 (0)	1 (0)	1 (0)	64.3
Ocean	1 (0)	2 (2.3%)	1 (0)	1 (0)	13.2	2 (2.3%)	2 (2.3%)	1 (0)	85.2
Barnes.	1 (0)	1 (0)	1 (0)	1 (0)	44.5	1 (0)	1 (0)	1 (0)	50.2
Cholesky∗	1 (0)	1 (0)	1 (0)	1 (0)	36.0	2 (16.7%)	2 (16.7%)	2 (25%)	52.2
Raytrace	1 (0)	1 (0)	1 (0)	1 (0)	32.5	1 (0)	2 (5.6%)	2 (16.7%)	59.3
Water	1 (0)	2 (5.9%)	1 (0)	1 (0)	39.8	2 (5.9%)	2 (5.9%)	2(16.7%)	62.0
MySQL1	2 (0.5%)	2 (0.5%)	1 (0)	2 (0.5%)	289.3	2 (0.5%)	3 (0.9%)	1 (0)	783.2
MySQL2	1 (0)	3 (2.4%)	1 (0)	3 (2.4%)	288.4	2 (1.2%)	3 (2.4%)	2 (3.5%)	892.2
MySQL3	1 (0)	1 (0)	1 (0)	1 (0)	281.5	1 (0)	1 (0)	1 (0)	832.2
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
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
MySQL5	1 (0)	1 (0)	1 (0)	1 (0)	224.5	2 (1%)	2 (1%)	1 (0)	800.3
MySQL6	2 (1%)	3 (1.9%)	1 (0)	3 (1.9%)	221.4	3 (1.9%)	4 (2.9%)	1 (0)	462.5
MySQL7	2 (0.9%)	2 (0.9%)	1 (0)	2 (0.9%)	312.5	3 (1.8%)	3 (1.8%)	1 (0)	1198.4
MySQL8	3 (2%)	3 (2%)	2 (2.2%)	3 (2%)	256.4	3 (2%)	3 (2%)	2 (2.2%)	999.5
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
Firefox1	1 (0)	1 (0)	1 (0)	1 (0)	239.4	2 (2.2%)	2 (2.2%)	1 (0)	618.5
Firefox2	2 (2.5%)	3 (5%)	1 (0)	3 (5%)	198.5	2 (2.5%)	3 (5%)	1 (0)	600.3
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
Bitcoin1	1 (0)	1 (0)	1 (0)	1 (0)	252.5	2 (1%)	2 (1%)	1 (0)	481.4
Bitcoin2	1 (0)	1 (0)	1 (0)	1 (0)	269.4	1 (0)	1 (0)	1 (0)	526.2
Bitcoin3	3 (0.9%)	3 (0.9%)	1 (0)	3 (1%)	302.8	4 (1.3%)	5 (1.3%)	1 (0)	558.8
Bitcoin4	2 (1.2%)	3 (2.4%)	1 (0)	2 (2.4%)	243.2	3 (2.4%)	3 (2.4%)	1 (0)	490.6

5.1.1 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 4). Without this step, SyncProf was less effective for 16 out of 24 bottlenecks using the CPWT metric, showing that the test selection step is beneficial.

5.2 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 first run DRD on all test cases ("DRD w/o selection" of Table 5). To evaluate SyncProf's ability in detecting bottlenecks regardless the quality of test cases, we also run only the selected bottleneck-exposing test cases on DRD ("DRD w/ selection" of Table 5). For each method, we compute the number N of CSs to

Table 5 DRD's effectiveness and efficiency in localizing bottlenecks

Bottleneck	DRD w/o selection			Time	DRD w/ selection			Time
	MAX	AVG	Conf. Int.	(min)	MAX	AVG	Conf. Int.	(min)
UTS	3	1.3	[1.2, 2.4]	22.5	2	1.2	[1.1, 2.2]	14.3
Radio.	3	1.5	[1.1, 2.5]	20.8	3	1.5	[1.1, 2.5]	12.2
Ocean	4	2.5	[1.8, 3.2]	30.3	3	2.0	[1.8, 2.9]	19.4
Barnes.	5	3.2	[2.1, 3.6]	19.3	4	2.8	[1.9, 3.2]	10.4
Cholesky∗	3	1.3	[1.2, 2.1]	15.3	3	1.1	[1.0, 1.5]	9.8
Raytrace	4	1.4	[1.2, 2.9]	16.8	2	1.2	[1.1, 2.4]	10.4
Water	3	1.5	[1.3, 3.5]	18.2	3	1.3	[1.2, 2.8]	11.5
MySQL1	7	3.6	[2.9, 5.2]	290.0	5	3.1	[2.5, 4.8]	89.8
MySQL2	11	6.8	[4.9,9.2]	282.4	8	5.2	[3.5, 8.8]	92.5
MySQL3	5	2.5	[2.8, 4.8]	285.3	5	1.5	[1.9, 2.9]	82.2
MySQL4-1	11	4.3	[2.9, 9.8]	202.7	8	4.3	[2.7, 9.2]	60.5
MySQL4-2	11	8.1	[6.4, 9.9]	202.7	9	8.1	[5.8, 8.5]	60.5
MySQL5	5	2.9	[2.5, 3.4]	198.6	5	2.5	[2.3, 3.2]	52.2
MySQL6	12	5.2	[3.7, 9.4]	208.7	9	4.5	[3.2, 8.1]	101.4
MySQL7	8	3.3	[2.2, 5.3]	300.0	8	3.0	[2.2, 4.5]	61.2
MySQL8	14	8.2	[5.5, 11.8]	228.4	12	7.5	[5.5, 9.9]	59.4
MySQL9∗	14	8.2	[5.5,11.8]	320.8	13	8.0	[5.3, 10.2]	82.2
Firefox1	6	3.0	[1.8, 5.4]	180.3	6	2.8	[1.8, 4.6]	62.9
Firefox2	13	9.1	[6.4, 10.2]	182.5	11	8.0	[6.2, 8.4]	70.2
Firefox3	13	8.3	[6.4, 10.2]	169.3	11	7.9	[6.1, 8.2]	55.9
Bitcoin1	3	2.8	[1.2, 2.6]	283.2	3	2.4	[1.2, 2.3]	189.2
Bitcoin2	3	1.3	[1.1, 2.5]	257.3	3	1.2	[1.1, 2.2]	115.5
Bitcoin3	6	4.2	[3.2, 5.1]	266.8	6	3.8	[3.2, 4.9]	129.5
Bitcoin4	7	4.5	[3.5, 5.5]	222.4	7	4.2	[3.3, 4.8]	120.1

inspect before hitting the desired CS for each run. Columns "MAX" and "AVG" of Table 4 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 with or without test selection.

For example, for MySQL2, SyncProf reports the desired CS at rank one or three (depending on the metric), whereas DRD without test selection is likely to rank the CS at a position between 4.9 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 without test selection, and compute the geometric mean across all bottlenecks. We find that, overall, SyncProf reduced the number of CSs to

inspect by an average of 55%, i.e., the approach almost halves the effort that a developer must spend. With test selection enabled, DRD was more effective in 13 out of 24 bottlenecks in terms number of CSs to inspect than that without test selection. Nevertheless, it required inspecting 42% more CSs than SyncProf on average.

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.

5.3 RQ3: effectiveness in suggesting optimizations

Table 6 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. For example, "-, (3)" means pattern 3 is detected and no optimizations are 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

Table 6 Effectiveness in suggesting optimizations

Bottleneck	Suggested optimization	Ground truth	Imp.
UTS	`--, (3)`	`nonblocking queue`	17%
Radio.	`--, (3)`	`nonblocking queue`	20%
Ocean	`CS reduction, (4)`	✓	22%
Barnes.	`--, (3)`	`increase lock array size`	21%
Cholesky.∗	`--, (3)`	`nonblocking queue`	19%
Raytrace	`--, (3)`	`nonblocking queue`	15%
Water	`CS reduction, (4)`	✓	12%
MySQL1	`--, --`	`replace a random generator`	80%
MySQL2	`lock split, (1,2)`	✓	160%
MySQL3	`lock elimination, (2)`	✓	125%
MySQL4-1	`lock split, (1,2), CS reduction, (4)`	✓	140%
MySQL4-2	`use reader locks, (2)`	✓	21%
MySQL5	`lock split, (1,2)`	✓	40%
MySQL6	~~`lock 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%
Bitcoin1	`CS reduction, (4)`	✓	92%
Bitcoin2	`lock elimination, (2)`	✓	72%
Bitcoin3	`lock elimination, (2)`	✓	44%
Bitcoin4	`--,- -`	`modify semantics of CSs`	39%

with the real fix. The last column lists the performance improvements after applying the optimizations. The improvement is calculated by averaging the improvements across all bottleneck-exposing test cases. The improvement of an execution is $\frac{T-T'}{T}$, where T is the original execution time and T' is the execution time after applying the optimization. To mitigate non-determinism, we ran each test case ten times.

As Table 6 shows, SyncProf finds optimization patterns in 18 out of 24 bottlenecks. For the 18 optimizable bottlenecks, SyncProf suggests 13 concrete optimizations; seven of them matched the ground truth and were beneficial. Applying the optimizations led to improvements between 12% and 160%. We further explain these results in four categories.

Patterns found and true optimizations suggested For twelve 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 be split into four different mutex objects. In Firefox1, the `jemalloc` function (an implementation of memory allocation) is protected using the same lock on both the timer thread the main thread for allocating objects. SyncProf suggests to use separate locks, as the two CSs do not conflict.

In Ocean, Water, MySQL4-1, and Bitcoin1, SyncProf detected the *long-critical-section* pattern and suggests to reduce the CS size. For example, in Ocean, the optimization is to reduce the amount of code in the CS to eliminate unnecessary lock acquire attempts (Heinrich and Chaudhuri 2003). In the Water program, the `InterfBar` barrier is used to synchronize updates of the `force` array. This barrier region is unnecessarily long and is suggested to be reduced. In Bitcoin1, the `SendMessages` function spends much time acquiring a `cs_main` lock to enter the CS. SyncProf suggests to reduce the size of this CS by moving a part of the code out of the CS. MySQL4-1 is discussed in Section 5.5.

Patterns found but optimizations not suggested For seven bottlenecks, SyncProf detected optimization patterns involving *low-degree conflicts* but does not suggest optimizations targeting specific CSs. For example, on UTS, Radiosity, Cholesky and Raytrace, 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 size (Woo et al. 1995). 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 woken 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, such that each lock is indexed in different sessions. Again, these optimizations require the developer's knowledge.

Patterns found and optimizations 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 suggest 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 three bottlenecks, SyncProf could not match any optimization patterns. On MySQL1, the random number generator inside the CS should be replaced with a lower cost one. On MySQL8, 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 wrzite functions execute concurrently. On Bitcoin4, the bottleneck occurs due the contention between the `GetTransaction` and the `ReadBlockFromDisk` functions. The real solution is to remove and rewrite parts of the CS code to reduce the CS size.

All of the above three 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.

5.4 RQ4: efficiency of SyncProf

The "Time" columns in Table 4 (shown previously) report the end-to-end total analysis time of SyncProf (including test selection, SyDG construction, performance impact analysis, USyDG construction, and optimization) and SyncProf without the first step that selects test cases. Specifically, the time spent on test case selection accounted for less than 2% of the overall testing time for each subject program. Without the first step of the approach, the analysis time 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 "Time" columns in Table 5 report the analysis time of DRD and DRD without the first step that selects test cases. Comparing to SyncProf, DRD is 8.7% more efficient in terms of the total execution time across all subject programs. This is 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 5.2).

The runtime overhead for binary instrumentation was about 10x for the 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 (Gong et al. 2015; Nistor et al. 2013). We consider these overheads to be acceptable for in-house performance profiling, which is the intended usage scenario of SyncProf.

5.5 A case study

We present an example (MySQL4-1 issue 75534) where SyncProf succeeded in detecting, localizing, and optimizing synchronization performance bottlenecks. InnoDB is the default

MySQL storage engine and maintains a storage area called the buffer pool for caching data and indexes in memory. When the InnoDB buffer pool is large, many data requests can be satisfied by retrieving the data from memory. However, in MySQL-4, bottlenecks occurred when multiple threads try to access the buffer pool at the same time. Figure 9 shows a code snippet where the bottleneck exists. The numbers on the left indicate the static CS IDs. Thread $T1$ calls the `btr_search_validate` to validate the search of InnoDB buffer pool. Threads $T2$ and $T3$ are two instances of the function `buf_page_get_gen` to get access to a database page and obtain pages and load them into the buffer. Thread $T4$ calls `innodb_buffer_pool_size_update` to resize the pages. If the request aims to reduce the buffer pool size, the `buf_pool_withdraw_blocks` function is invoked.

Figure 10 shows a partial SyDG from a MySQL execution trace. Each node indicates a CS with an object. The nodes $< 4, 5, 6 >$, $< 3, 7, 8 >$ and $< 11, 12 >$ involve nested critical sections. When applying SyncProf to the program using the APLT metric, node 3 has the highest performance impact, followed by nodes 12 and 4.

Figure 11 shows a partial USyDG. The edges (T1:1, T2:3), (T2:3, T3:9) and (T3:3, T4:10) are removed according to the *read-read* pattern. Since the CSs T2:5 and T2:6 are nested inside the CS T2: 4, the union of their lock set is assigned to the CS T1:2. SyncProf splits the `buff` lock on the CSs T2:3, T3:3, T3:9 and T4:10 into two different locks (i.e., `LRU` and `list`). Furthermore, the sizes of the CSs T3:7 and T4:12 are reduced because the *long-critical-section* pattern is detected. These optimizations are correct and have been known before (MySQL4-1 in Table 6).

```
btr_search_validate(void) {
   ...
1. buf_pool_mutex_enter(buf_pool);
   cell_count = hash_get_n_cells(...);
   ...
   buf_pool_mutex_exit(buf_pool);
   ...
2. buf_pool_mutex_enter(buf_pool);
   ...
   buf_pool_mutex_exit(buf_pool);
}
```

```
buf_pool_withdraw_blocks(...){
   ...
   if (...){
10.   buf_pool_mutex_enter(buf_pool);
      scan_depth = UT_LIST_GET_LEN(
           buf_pool->LRU);
      buf_pool_mutex_exit(buf_pool);
   }
   ...
11.buf_pool_mutex_enter(buf_pool);
   ...
   while (bpage != NULL) {
   ...
12.   mutex_enter(block_mutex);
      ...
      mutex_exit(block_mutex);
      ...
      buf_pool_mutex_exit(buf_pool);
}
```

```
buf_page_get_gen(...){

   buf_page_init_for_read (...);
   ...
4. buf_pool_mutex_enter(buf_pool);
   ...
   if (...) {
5.   buf_page_mutex_enter(block);
     ...
6.   mutex_enter(&buf_pool->zip_mutex);
     fix_block = block;
     buf_LRU_block_free_non_file_page(
          block);
     ...
     mutex_exit(&buf_pool->zip_mutex);
     ...
     buf_page_mutex_exit(block);
     ...
   }
   buf_pool_mutex_exit(buf_pool);
   access_time = buf_page_is_accessed(&
        block->page);

buf_page_init_for_read(...) {
   ...
3. buf_pool_mutex_enter(buf_pool);
   hash_lock = buf_page_hash_lock_get
        (...);
   if (...){
7.    buf_page_mutex_enter(block);
      buf_LRU_block_free_non_file_page(
           block);
      buf_page_mutex_exit(block);
   }
   if (...) {
8.    mutex_enter(&buf_pool->zip_mutex)
      ;
       ...
      mutex_exit(&buf_pool->zip_mutex);
   }
   ...
   buf_pool_mutex_exit(buf_pool);
}
```

Fig. 9 Code snippet showing the synchronization bottleneck of MySQL-4

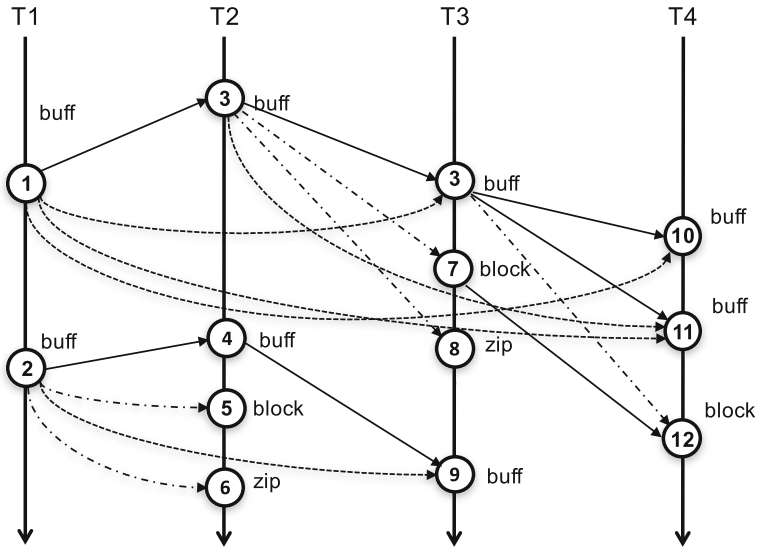


Fig. 10 A SyDG for MySQL issue 75534

6 Discussion

It is possible that optimizing a CS introduces a new bottleneck due to CS reordering. We address this problem by validating 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 3.2.2).

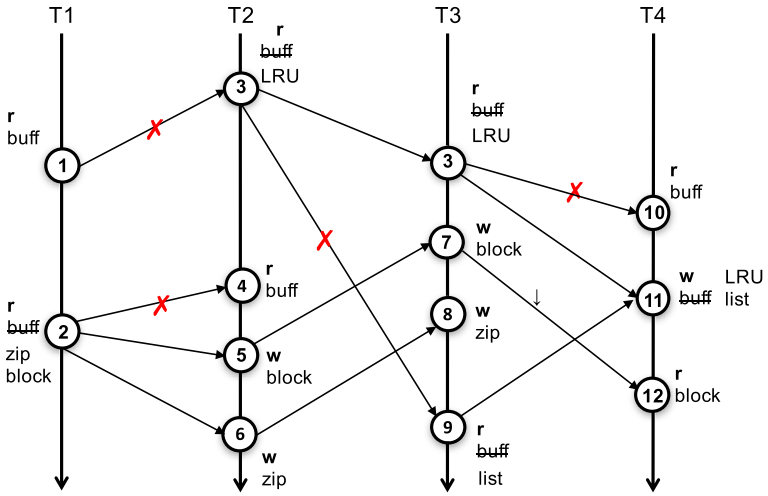


Fig. 11 An USyDG for MySQL

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 affect the program's 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.

It is also possible that the amount of work performed inside each CS can vary significantly over time. For example, one queue can be shrinking at one point but growing at a later point. As such, the performance impact of a highly contended CS at a certain point can be amortized by the rest of its executions. One possible solution is to add hardware support in the deployed environment to adaptively accelerate the program execution (Bois et al. 2013; Joao et al. 2012).

The test cases that are selected in the first step may affect the effectiveness of synchronization bottleneck detection. We next examine the influence of the CPU usage and workload parameters used for test case selection.

CPU usage While longer execution times and low CPU usage are necessary conditions of synchronization performance bottlenecks, we believe that exercising the test cases used by SyncProf may also trigger other non-CPU activities, such as I/O intensive code, leading to low CPU usage. Therefore, we further assess the effects of choosing different levels of CPU usage. In addition to the default CPU usage (90%), we studied two other different thresholds: 70% and 100%. When the threshold is set to 70%, a test case is only selected when the mean of $\mathcal{U}'$ is less than 70%. When the threshold is set to 100%, it indicates that only execution time is considered when selecting test cases (i.e., the first two conditions described Section 3.1)

In Table 7, Columns 2-7 list the number of selected test cases and the effectiveness scores of the CPWT metric under the default setting, 70% CPU usage and the 100% CPU usage, respectively. Here we choose CPWT because it performs best among all metrics. When the threshold is 70%, 12 out of 25 bottlenecks cannot be detected because none of the test cases can bring the CPU usage down to 70% under the given workloads. When the threshold is set to 100%, comparing to the default setting, more test cases are selected. The effectiveness is decreased on eight programs (rendered in bold font). These results indicate that using a high CPU threshold can help select test cases that generally affect the system's performance, but these test cases may not be specific to exposing synchronization performance bottlenecks. On the other hand, a low CPU threshold may overconstrain the test selection and adversely affect the bottleneck detection effectiveness. Therefore, the CPU threshold must be tuned into an appropriate value, and in our case the optimal threshold is 90%.

Workload increments The default number of increments of the workload is set to 50, i.e., $\delta_{max} = 50$ (Section 3.1). However, if δ_{max} is too small, it may discard test cases that could have been selected otherwise. For instance, in a file archiver application (e.g., gzip), while increasing the file size increases the workload, it also spawns more threads, which is used to speedup the execution time. As such, increasing the workload in a small scale does not increase the execution time or CPU usage. The workload must reach a certain value to expose synchronization performance bottlenecks.

Here, we examine how varying δ_{max} can affect the effectiveness and efficiency of SyncProf. We studied $\delta_{max} = 20$ and $\delta_{max} = 80$. The results are shown in the Columns 8-11 of Table 7. On five out of the 25 programs, when $\delta_{max} = 20$, SyncProf fails to select test cases due to the small workload size. On the other 20 programs, SyncProf achieves

Table 7 Effectiveness when using different CPU usage thresholds and number of increments of workloads

Bottleneck	$\overline{\mathcal{U}}' < 90\%$, $\delta_{max} = 50$		$\overline{\mathcal{U}}' < 70\%$, $\delta_{max} = 50$		$\overline{\mathcal{U}}' < 100\%$, $\delta_{max} = 50$		$\overline{\mathcal{U}}' < 90\%$, $\delta_{max} = 20$		$\overline{\mathcal{U}}' < 90\%$, $\delta_{max} = 80$	
	T_{sel}	CPWT	T_{sel}	CPWT	T_{sel}	CPWT	T_{sel}	CPWT	T_{sel}	CPWT
UTS	8	1 (0)	0	–	10	**2 (0.9%)**	0	–	**10**	1 (0)
Radio.	14	1 (0)	6	1 (0)	20	1 (0)	14	1 (0)	**16**	1 (0)
Ocean	21	1 (0)	10	1 (0)	32	1 (0)	15	1 (0)	21	1 (0)
Barnes.	9	1 (0)	0	–	19	1 (0)	0	–	9	1 (0)
Cholesky∗	12	1 (0)	5	1 (0)	15	**2 (16.7%)**	6	1 (0)	12	1 (0)
Raytrace	12	1 (0)	5	1 (0)	15	1 (0)	5	1 (0)	**15**	1 (0)
Water	9	1 (0)	4	1 (0)	16	1 (0)	4	1 (0)	12	1 (0)
MySQL1	31	2 (0.5%)	10	2 (0.5%)	65	2 (0.5%)	18	2 (0.5%)	31	2 (0.5%)
MySQL2	29	1 (0)	0	–	64	**2 (1.2%)**	22	1 (0)	**38**	1 (0)
MySQL3	32	1 (0)	11	1 (0)	65	1 (0)	25	1 (0)	**37**	1 (0)
MySQL4-1	32	2 (0.8%)	0	–	71	2 (0.8%)	0	–	32	2 (0.8%)
MySQL4-2	21	5 (3.3%)	0	–	49	**6 (4.1%)**	0	-	**29**	5 (3.3%)
MySQL5	24	1 (0)	35	1 (0)	49	1 (0)	19	1 (0)	24	1 (0)
MySQL6	43	2 (1%)	0	–	52	2 (1%)	0	-	43	2 (1%)
MySQL7	25	2 (0.9%)	0	–	58	**3 (1.8%)**	11	2 (0.9%)	**32**	2 (0.9%)
MySQL8	25	3 (2%)	0	–	42	3 (2%)	20	3 (2.8%)	**30**	3 (2%)
MySQL9∗	25	3 (1.8%)	12	3 (1.8%)	61	3 (1.8%)	22	3 (1.8%)	25	3 (1.8%)
Firefox1	15	1 (0)	13	1 (0)	31	1 (0)	10	1 (0)	15	1 (0)
Firefox2	11	2 (2.5%)	0	–	28	2 (2.5%)	8	2 (2.5%)	11	2 (2.5%)
Firefox3	12	3 (2.2%)	0	-	26	**5 (4.4%)**	10	3 (2.2%)	**19**	3 (2.2%)
Bitcoin1	61	1 (0)	42	1 (0)	69	1 (0)	61	1 (0)	61	1 (0)
Bitcoin2	30	1 (0)	15	1 (0)	71	1 (0)	22	1 (0)	30	1 (0)
Bitcoin3	40	3 (0.9%)	0	–	72	**4 (1.3%)**	35	3 (0.9%)	40	3 (0.9%)
Bitcoin4	39	2 (1.2%)	0	–	65	**3 (2.4%)**	18	2 (1.2%)	39	2 (1.2%)

the same effectiveness as the default setting but requires fewer test cases. When $\delta_{max} = 80$, SyncProf detects all bottlenecks that are detected using the default setting. However, on 7 out of the 25 programs, it uses more test cases to expose the bottlenecks (rendered in bold font). These results indicate that the workload size does affect the effectiveness and efficiency of SyncProf, and that choosing appropriate workload sizes is important. The default value $\delta_{max} = 50$ is more cost-effective than the other values.

7 Related work

Several profiling techniques identify and optimize synchronization bottlenecks using software and hardware approaches (Tallent et al. 2010; Eyerman and Eeckhout 2010; Chen and Stenstrom 2012; Joao et al. 2012; Lozi et al. 2012; Identify Thread Contention 2015; Miller et al. 1990). These techniques focus on individual executions. For example,

Tallent et al. (2010) use an idleness metric to locate the threads that are responsible for the idleness, so the threads can be accelerated by the hardware. This approach does not pinpoint the root cause of synchronization bottlenecks. Joao et al. (2012) propose a cooperative software-hardware approach to identify and accelerate the most critical serializing bottlenecks at runtime by counting the number of waiters on each CS. Chen and Stenstrom (2012) use critical path analysis to identify performance bottlenecks in multithreaded applications. The foregoing techniques and our technique both require performance metrics to identify bottlenecks. Dynatrace (Identify Thread Contention 2015) and Intel Vtune (Intel® vtune™ amplifier xe 2014) report lock contention by computing thread synchronization time. Similar to DRD, they analyze individual executions, 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.

Moreover, few existing tools on profiling synchronization bottlenecks can localize the root cause of synchronization bottleneck in the source code or suggest optimization opportunites. For example, DYNATRACE (Identify Thread Contention 2015) can determine whether performance slowdown is caused by synchronization at a high-level, but does not point to the specific code.

Other research finds performance problems through dynamic analysis (Selakovic et al. 2017; Han et al. 2012; Nistor et al. 2013; Jin et al. 2012; Ammons et al. 2004) or static pattern matching (Selakovic and Pradel 2016). For example, MemoizeIt (Toffola et al. 2015) detects repeated method calls that can be optimized through memoization. StackMine mines call stack traces to discover call sequences with a high performance impact (Han et al. 2012). It then applies a clustering algorithm to group similar callstack patterns that lead to the performance problem (Han et al. 2012). While the above techniques are inspiring and effective, they focus on sequential programs.

Several approaches analyze concurrency-related performance issues (Pradel et al. 2014; Yu et al. 2014; Wert et al. 2013; Gu et al. 2015; Curtsinger and Berger 2015; Alam et al. 2017). For example, SpeedGun (Pradel et al. 2014) generates multi-threaded performance test cases to expose performance differences between two program versions. Yu et al. (2014) propose a trace-based dynamic approach to effectively identify general performance problems (including lock contention) and report root causes. Wert et al. (2013) 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 bottlenecks or suggests optimizations. Alam et al. (Alam et al. 2017) et al. develop a lightweight approach to detect synchronization-related performance bottlenecks, but their approach is based on single executions and does not suggest optimizations.

Some techniques optimize critical sections or locks to improve performance (Zheng et al. 2015; Curtsinger and Berger 2015; Sim et al. 2012; Rajwar and Goodman 2001; Roy et al. 2009). For example, Lock Elision (LE) (Rajwar and Goodman 2001; Roy et al. 2009) uses a hardware approach to dynamically remove unnecessary locks. However, this approach does not localize bottlenecks or suggest optimizations at the code level. Zheng et al. (2015) identify unnecessary locks from single executions. This approach, however, does not localize the root cause of synchronization bottlenecks. Curtsinger and Berger (2015) use a causal profiling approach to identify code with optimization opportunities. Their approach 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 (Koskinen and Herlihy 2008; Yu et al. 2014; Wang et al. 2008; Gray et al. 1975; Samak and Ramanathan 2014). For example, the wait-for graph and its extensions have been widely used to detect deadlocks (Koskinen and Herlihy 2008; Wang et al. 2008; Samak and Ramanathan 2014). Yu et al. (2014) 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.

8 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. In the context of automated program repair, SyncProf contributes an approach that supports developers in repairing programming mistakes beyond functional correctness, in particular, performance bottlenecks.

Acknowledgments This research is supported in part by the NSF grants CCF-1464032 and CCF-1652149, 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".

References

Alam K, Ahmad R, Ko K (2017) Enabling far-edge analytics: performance profiling of frequent pattern mining algorithms. IEEE Access

Ammons G, Choi J-D, Gupta M, Swamy N (2004) Finding and removing performance bottlenecks in large systems

Arlitt MF, Williamson CL (1996) Web server workload characterization: the search for invariants. 126–137

Artho C, Havelund K, Biere A (2003) High-level data races. J Softw Test Verif Reliab 13:207–227

Avritzer A, Kondek J, Liu D, Weyuker EJ (2002) Software performance testing based on workload characterization. In: Proceedings of the international workshop on software and performance, pp 17–24

Barford P, Crovella M (2001) Critical path analysis of TCP, transactions. Proc Conf Appl Technol Architect Protoc Comput Commun 9:238–248

Bois K, Eyerman S, Sartor JB, Eeckhout L (2013) Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In: Proceedings of the 40th annual international symposium on computer architecture, pp 511–522

Bond MD, Coons KE, McKinley KS (2010) PACER: proportional detection of data races. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 255–268

Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S (2010) A randomized scheduler with probabilistic guarantees of finding bugs. In: Proceedings of the international conference on architectural support for programming languages and operating systems, pp 167–178

Chen G, Stenstrom P (2012) Critical lock analysis: Diagnosing critical section bottlenecks in multi-threaded applications. In: Proceedings of the international conference on high performance computing, networking, storage and analysis

Choudhary A, Lu S, Pradel M (2017) Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: International conference on software engineering, pp 266–277

Cleve H, Zeller A (2005) Locating causes of program failures. In: Proceedings of the international conference on software engineering, pp 342–351

Coons KE, Burckhardt S, Musuvathi M (2010) GAMBIT: effective unit testing for concurrency libraries. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, pp 15–24

Curtsinger C, Berger ED (2015) Coz: finding code that counts with causal profiling. In: Proceedings of the ACM symposium on operating systems principles, pp 184–197

Diagnosing Lock Contention with the Concurrency Visualizer (2010) Microsoft MSDN

Draheim D, Grundy J, Hosking J, Lutteroth C, Weber G (2006) Realistic load testing of web applications. In: Conference on software maintenance and reengineering, pp 11–70

DRD (2015) A thread error detector, http://valgrind.org/docs/manual/drd-manual.html

Identify Thread Contention (2015) https://community.dynatrace.com

Intel® vtune™ amplifier xe (2014) http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

Effinger-Dean L, Lucia B, Ceze L, Grossman D, Boehm H-J (2012) Ifrit: interference-free regions for dynamic data-race detection. In: Proceedings of the ACM SIGPLAN international conference on object oriented programming systems languages and applications, pp 467–484

Eyerman S, Eeckhout L (2010) Modeling critical sections in amdahl's law and its implications for multicore design. In: Proceedings of the international symposium on computer architecture, pp 362–370

Flanagan C, Freund SN (2004) Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the international symposium on principles of programming languages, pp 256–267

Flanagan C, Qadeer S (2003) A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 338–349

Fouché S, Cohen MB, Porter A (2007) Towards incremental adaptive covering arrays. In: Proceedings of the ACM SIGSOFT symposium on foundations of software engineering, pp 557–560

Gong L, Pradel M, Sen K (2015) JITProf pinpointing JIT-unfriendly JavaScript code. In: Proceedings of the ACM SIGSOFT symposium on foundations of software engineering, pp 357–368

Gray JN, Lorie RA, Putzolu GR (1975) Granularity of locks in a shared data base. In: Proceedings of the international conference on very large data bases, pp 428–451

Gu R, Jin G, Song L, Zhu L, Lu S (2015) What change history tells us about thread synchronization. In: Proceedings of the ACM SIGSOFT symposium on foundations of software engineering, pp 426–438

Gupta R, Epstein M (1990) Achieving low cost synchronization in a multiprocessor system. Fut Gen Comput Syst 6(3):255–269

Han S, Dang Y, Ge S, Zhang D, Xie T (2012) Performance debugging in the large via mining millions of stack traces. In: Proceedings of the international conference on software engineering, pp 145–155

Heinrich M, Chaudhuri M (2003) Ocean warning: avoid drowning. SIGARCH Comput Archit News 31(3):30–32

Heirman W, Carlson T, Che S, Skadron K, Eeckhout L (2011) Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In: International symposium on workload characterization, pp 38–49

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 77–88

Joao JA, Suleman MA, Mutlu O, Patt YN (2012) Bottleneck identification and scheduling in multithreaded applications. In: Proceedings of the international conference on architectural support for programming languages and operating systems, pp 223–234

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of international conference on automated software engineering, pp 273–282

Joshi S, Lahiri SK, Lal A (2012) Underspecified harnesses and interleaved bugs. In: Proceedings of the international symposium on principles of programming languages, pp 19–30

Kahlon V, Sinha N, Kruus E, Zhang Y (2009) Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the ACM SIGSOFT symposium on foundations of software engineering, pp 13–22

Koskinen E, Herlihy M (2008) Dreadlocks: efficient deadlock detection. In: Proceedings of the symposium on parallelism in algorithms and architectures, pp 297–303

Lozi J-P, David F, Thomas G, Lawall J, Muller G (2012) Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: USENIX annual technical conference, pp 6–6

Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 190–200

Marino D, Musuvathi M, Narayanasamy S (2009) LiteRace: effective sampling for lightweight data-race detection. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 134–143

Mertler CA, Vannatta RA (2002) Advanced and multivariate statistical methods, Pyrczak, Los Angeles

Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the ACM symposium on principles of distributed computing, pp 267–275

Miller B, Clark M, Hollingsworth J, Kierstead S, Lim S-S, Torzewski T (1990) IPS-2: the second generation of a parallel program measurement system. In: IEEE transactions on parallel and distributed systems, pp 206–217

Mosberger D, Jin T (1998) A tool for measuring web server performance. ACM SIGMETRICS Perform Eval Rev 26(3):31–37

Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I (2008) Finding and reproducing Heisenbugs in concurrent programs. In: Proceedings of the USENIX conference on operating systems design and implementation, pp 267–280

Naik M, Park C-S, Sen K, Gay D (2009) Effective static deadlock detection. In: Proceedings of the international conference on software engineering, pp 386–396

Nistor A, Luo Q, Pradel M, Gross TR, Marinov D (2012) Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code. In: Proceedings of the international conference on software engineering, pp 727–737

Nistor A, Song L, Marinov D, Lu S (2013) Toddler: detecting performance problems via similar memory-access patterns. In: Proceedings of the international conference on software engineering, pp 562–571

Novillo E, Lu P (2003) A case study of selected SPLASH-2 applications and the sbt debugging tool. In: Proceedings of the international symposium on parallel and distributed processing, p 290.2

Olivier S, Huan J, Liu J, Prins J, Dinan J, Sadayappan P, Tseng C-W (2007) UTS: an unbalanced tree search benchmark. In: LCPC, pp 235–250

Ongoing work on lock contention in QEMU driver (2013). https://www.redhat.com/archives/libvir-list/2013-May/msg01247.html

Pradel M, Gross TR (2012) Fully automatic and precise detection of thread safety violations. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 521–530

Pradel M, Huggler M, Gross TR (2014) Performance regression testing of concurrent classes. In: Proceedings of the international symposium on software testing and analysis, pp 13–25

Pradel M, Schuh P, Necula G, Sen K (2014) EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In: Proceedings of the ACM SIGPLAN international conference on object oriented programming systems languages and applications

Rajwar R, Goodman JR (2001) Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the ACM/IEEE international symposium on microarchitecture, pp 294–305

Roy A, Hand S, Harris T (2009) A runtime system for software lock elision. In: Proceedings of the SIGOPS/EuroSys European conference on computer systems, pp 261–274

Sahelices B, Ibáñez P, Viñals V, Llabería JM (2009) A methodology to characterize critical section bottlenecks in dsm multiprocessors. In: Proceedings of the international euro-par conference on parallel processing, pp 149–161

Samak M, Ramanathan MK (2014) Trace driven dynamic deadlock detection and reproduction. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, pp 29–42

Selakovic M, Pradel M (2016) Performance issues and optimizations in JavaScript: an empirical study. In: Proceedings of the international conference on software engineering

Selakovic M, Glaser T, Pradel M (2017) An actionable performance profiler for optimizing the order of evaluations. In: International symposium on software testing and analysis, pp 170–180

Sen K (2007) Effective random testing of concurrent programs. In: Proceedings of international conference on automated software engineering, pp 323–332

Sen K (2008) Race directed random testing of concurrent programs. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 11–21

Shacham O, Bronson N, Aiken A, Sagiv M, Vechev M, Yahav E (2011) Testing atomicity of composed concurrent operations. In: Proceedings of the ACM SIGPLAN international conference on object oriented programming systems languages and applications, pp 51–64

Sim J, Dasgupta A, Kim H, Vuduc R (2012) A performance analysis framework for identifying potential benefits in GPGPU applications. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 11–22

Tallent NR, Mellor-Crummey JM, Porterfield A (2010) Analyzing lock contention in multithreaded applications. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, pp 269–280

Toffola LD, Pradel M, Gross TR (2015) Performance problems you can fix: a dynamic analysis of memoization opportunities. In: Proceedings of the ACM SIGPLAN international conference on object oriented programming systems languages and applications

Visser W, Havelund K, Brat GP, Park S, Lerda F (2003) Model checking programs. Autom Softw Eng 10(2):203–232

von Praun C, Gross TR (2003) Static conflict analysis for multi-threaded object-oriented programs. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 115–128

Wang L, Stoller SD (2006) Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming, pp 137–146

Wang Y, Kelly T, Kudlur M, Lafortune S, Mahlke S (2008) Gadara: dynamic deadlock avoidance for multithreaded programs. In: Proceedings of the USENIX conference on operating systems design and implementation, pp 281–294

Wert A, Happe J, Happe L (2013) Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In: Proceedings of the international conference on software engineering, pp 552–561

Williams A, Thies W, Ernst MD (2005) Static deadlock detection for java libraries. In: European conference on object-oriented programming, pp 602–629

Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the international symposium on computer architecture, pp 24–36

Xu M, Bodík R, Hill MD (2005) A serializability violation detector for shared-memory server programs. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, pp 1–14

Yu T, Pradel M (2016) Syncprof: detecting, localizing, and optimizing synchronization bottlenecks. In: Proceedings of the international symposium on software testing and analysis, pp 389–400

Yu X, Han S, Zhang D, Xie T (2014) Comprehending performance from real-world execution traces: a device-driver case. In: Proceedings of the international conference on architectural support for programming languages and operating systems, pp 193–206

Zheng L, Liao X, He B, Wu S, Jin H (2015) On performance debugging of unnecessary lock contentions on multicore processors: a replay-based approach. In: Proceedings of the IEEE/ACM international symposium on code generation and optimization

Tingting Yu is an Assistant Professor of Computer Science at University of Kentucky. She received her M.S. and Ph.D degree from University of Nebraska-Lincoln in 2014, and B.E. degree in Software Engineering from Sichuan University in 2008. Her research is in software engineering, with focus on developing methods and tools for improving reliability and security of complex software systems; testing for sequential and concurrent software; regression testing; and performance testing.

Michael Pradel is an assistant professor at TU Darmstadt, which he joined after a PhD at ETH Zurich and a post-doc at UC Berkeley. His research interests span software engineering and programming languages, with a focus on tools and techniques for building reliable, efficient, and secure software. In particular, he is interested in dynamic program analysis, test generation, concurrency, performance profiling, and JavaScript-based web applications.