SherLock: Unsupervised Synchronization-Operation Inference

Guangpu Li
University of Chicago
USA
cstjygpl@uchicago.edu

Dongjie Chen
Nanjing University
China
dongjie@smail.nju.edu.cn

Shan Lu
University of Chicago
USA
shanlu@uchicago.edu

Madanlal Musuvathi
Microsoft Research
USA
madanm@microsoft.com

Suman Nath
Microsoft Research
USA
suman.nath@microsoft.com

ABSTRACT

Synchronizations are fundamental to the correctness and performance of concurrent software. Unfortunately, correctly identifying all synchronizations has become extremely difficult in modern software systems due to the various types of synchronizations. Previous work either only infers specific type of synchronization by code analysis or relies on manual effort to annotate the synchronization.

This paper proposes SherLock, a tool that uses unsupervised inference to identify synchronizations. SherLock leverages the fact that most synchronizations appear around the conflicting operations and form it into a linear system with a set of synchronization properties and hypotheses. To collect enough observations, SherLock runs the unit tests a small number of times with feedback-based delay injection.

We applied SherLock on 8 C# open-source applications. Without any prior knowledge, SherLock inferred 122 unique synchronizations, with few false positives. These inferred synchronizations cover a wide variety of types, including lock operations, fork-join operations, asynchronous operations, framework synchronization, and custom synchronization.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**;
• **Computer systems organization** → *Reliability*.

KEYWORDS

Synchronization, Unsupervised inference, Concurrency, Happens-before inducing

ACM Reference Format:
Guangpu Li, Dongjie Chen, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2021. SherLock: Unsupervised Synchronization-Operation Inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446754

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446754

1 INTRODUCTION

Synchronization operations are fundamental to the correctness and performance of concurrent software. They induce happens-before relationships, and determine which operations can or cannot execute concurrently. Many tools for analyzing concurrent programs, such as bug finding [9, 17, 20, 21, 25, 32, 34, 35, 38, 42, 46], bug fixing [22–24, 27, 33], performance profiling [3, 8, 13, 14, 29], and record-and-replay [37, 41] tools rely on understanding the semantics of program synchronizations and the happens-before relationship induced by them. Typically, these tools rely on manual specification of this semantics. Incorrect and/or incomplete synchronization specifications severely limits the effectiveness of concurrency tools.

Unfortunately, correctly identifying all happens-before-inducing synchronizations in modern software systems is a challenging task.

Simply identifying all low-level primitives like atomic operations or fences would not work. These operations do not always induce happens-before relationship, like when an atomic operation is used to increment a statistics variable. Even when they do, their synchronization is often induced upon library code, while their impact to application code is often unclear and requires manual specifications.

Identifying common threading and locking APIs, such as those provided by `pthread` APIs, is far from sufficient, as modern concurrent and distributed programs use various mechanisms to induce happens-before synchronization, including data-parallel processing primitives, event-based asynchronous operations, shared-memory flag variables, language enforced semantics (e.g., finalizers only execute after an object is unreachable), system calls, and even server-side synchronization. Moreover, each form of coordination often has many variations: for example, C# standard `threading` library offers 5 lock classes and 9 signal-wait classes, with each containing many synchronization APIs and many sub-classes [6]. To complicate things further, programs and frameworks can provide their own mechanisms for enforcing happens-before relationship in concurrent execution, without using traditional locks or signal/wait mechanisms [1]. These semantic-rich operations are often difficult to annotate, as we will see in Section 5.3.

All this complexity eliminates the hope of a once-for-all tedious manual annotations, which are likely to be error-prone anyway.

In this paper, we cast the synchronization inference as a *dynamic unsupervised* probabilistic inference problem. The basic idea is to

dynamically observe various signals of synchronization behaviors during representative executions (say, during testing). While each signal could be noisy, the goal is to cumulatively combine these signals over multiple executions to predict synchronizations with high confidence. Being unsupervised has the crucial advantage that one needs no user-provided annotations, which we believe is essential for the general applicability of this technique.

This paper is inspired by prior work on probabilistic inference for security specifications [12, 31], which identifies source, sink, and sanitizers for vulnerability detection. In contrast to our work, these works use a semi-supervised approach that requires manual annotations to bootstrap their analysis. Also, they analyze the programs statically, while a key hypothesis of our work is that dynamic program behavior provides us a variety of signals to identify synchronization whose precision cannot be matched by those available statically.

This paper proposes SherLock, a tool for identifying program synchronizations. At high level, SherLock is based on the idea that all synchronizations are used to order events that would otherwise result in bugs. For instance, a data race occurs when two threads concurrently access the same variable with at least one of them being a write, which we refer to as concurrent conflicting accesses. Consider two conflicting accesses a and b in a dynamic execution as shown in Figure 2.a with a occurring before b. To prevent them from becoming a data race, programmers need to enforce a *happens-before* [26] relation between them by using a pair of synchronizations: a *release* synchronization after a and an *acquire* synchronization before b. The acquire synchronization blocks the execution of b until the release (and thus a) is complete or changes the control flow to prevent the execution of b. In this paper, we define *synchronization* as any operation or instruction that participates in forming a happens-before relation across threads.

Specifically, SherLock identifies synchronizations by relying on the following three insights.

Insight 1. We hypothesize that most (if not all) such conflicting accesses in mature programs are properly synchronized. Thus, if one considers an execution in Figure 2.a, it is highly likely that one of the operations in the *release window* that follows a is a release synchronization and one operation in the *acquire* window that precedes b is an acquire synchronization.

Insight 2. While we may not be able to precisely identify from a single execution which of the operations in the release (or acquire) window offers a release (or acquire, respectively) synchronization, we can do so by observing multiple executions and considering other characteristics of synchronization. Specifically, we design a set of properties and hypotheses that reflect fundamental assumptions of synchronizations and their behavior. They work together to enable us to pinpoint synchronizations. We discuss the details in Section 2.

Insight 3. Effective inference depends on conflicting operations, such as the ones in Figure 2.a, being temporally close. Otherwise, large acquire/release windows will produce too many candidate operations. Rather than relying on the underlying scheduler to get lucky, we can actively perturb the execution at strategic locations to improve inference. We discuss the details in Section 3.

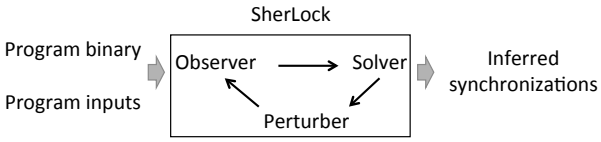

Figure 1: SherLock workflow

Guided by these insights, we have designed SherLock. As shown in Figure 1, given a program binary and its inputs, without any annotation, SherLock runs the program for all inputs and identifies a set of methods and variables whose entrances and/or exits, reads and/or writes are synchronizations, through three components working together:

1. Observer. This component instruments software binaries and produces execution traces under the provided inputs. The goal of Observer is to make relevant observation of software behavior that can be compared against those properties and hypotheses of synchronization operations by the Solver. Its detailed implementation will be introduced in Section 4.1.

2. Solver. It processes all the observations collected so far. It identifies a set of acquire synchronizations and a set of release synchronizations, that all strictly satisfy those synchronization properties and altogether violate those hypotheses about synchronizations the least. This procedure is conducted by first encoding all the observations into linear constraints, representing must-be-satisfied properties, and objective functions, representing better-be-satisfied hypotheses, and then using a linear solver [2] to output every operation's probability of being an acquire or a release synchronization. The details are presented Section 4.2.

3. Perturber. To help the observer to make useful observations, without fully relying on the random factors in concurrent program execution, the perturber injects delays at strategic points of the execution based on the Solver's earlier results. These delays will help observers collect key observations, which will then help the solver to draw more confident conclusions. The details will be presented in Section 3.

We applied SherLock on 8 C# open-source applications. In total, SherLock inferred 122 unique true synchronizations with few false positives. These include 1) standard synchronization primitives, like locks (`Monitor.Enter/Exit`), fork-join (`Task.Start/Wait`) and asynchronous tasks (`DataflowBlock.Post/Receive`); (2) variable based synchronizations such as spin loops and flag variables; and, (3) application-specific methods that enforce happens-before relations by relying on underlying frameworks and language semantics (e.g. order between last-reference-removing instruction and the object dispose). A version of FastTrack [17, 18] that we built for C# applications detects 7× more true data races and 8× fewer false data races by using these inferred synchronization than the default.

This paper makes the following contributions:

- Identifying a number of properties and hypotheses reflecting fundamental assumptions about synchronizations and how they are typically used in software, that work together to support effective synchronization inference.

- A feedback-based delay injection that actively exposes, instead of passively observes, run-time software behaviors that are particularly helpful for synchronization inference.
- An artifact SherLock that uses unsupervised inference to identify synchronizations with high coverage and accuracy.

The source code of SherLock is available at https://github.com/SpectrumLi/SherLock. A virtual machine image that can be used to easily reproduce all the experimental results in this paper is available and described in details in the Artifact Appendix.

2 WHAT ARE SYNCHRONIZATION BEHAVIORS?

To effectively pinpoint synchronizations, we will use a set of properties and hypotheses that capture common behaviors of synchronizations. The properties represent fundamental assumptions that every synchronization should satisfy; the hypotheses reflect how *most* synchronizations are *typically* designed and used. They work together to support our inference.

The goal of this paper is to identify synchronizations in the application *without* understanding the semantics of underlying framework, library, or operating system that implements them. As mentioned before, we define synchronization as any instruction or operation in the application that enforces a happens-before relation across threads. In this paper, we consider every synchronization, acquire or release, to take one of these forms: a read of a heap variable; a write to a heap variable; an invocation or entry point of an API or method; and an exit of an API or method. For example, the invocation of a thread-creation API and the entry point of the specified delegate method of the child thread form a pair of release and acquire that we aim to identify. Note that the actual implementation of the threading library or framework that enforces this happens-before relation is irrelevant to SherLock.

Property: Read-Acquire & Write-Release. Not every operation has the capability to release or acquire. Among memory-access operations, a heap read does not change system states and hence cannot be a release synchronization; on the other hand, a heap write cannot perceive what is going on in the system, and hence cannot be as an acquire synchronization. Similarly for method-related operations, the invocation of an application method can block the caller till some condition is satisfied, while the exit of an application method may satisfy such conditions. Consequently, we associate a method's exit with a release and a method's invocation as an acquire, but never the other way around. We also enforce that a release synchronization cannot be an acquire and vice versa.

In addition to these properties, we add a set of hypotheses which are *soft* constraints — they are satisfied by most synchronizations most of the time, but not always.

Hypothesis: Mostly Protected. Most, if not all, conflicting accesses in a mature software should be synchronized. Consequently, given the observation in Figure 2.a, we could hypothesize that there probably exists at least one release among a_1, a_2, and a_3, which form the *release window*; and that there probably exists at least one acquire among b_1, b_2, and b_3, which form the *acquire window*.

Hypothesis: Synchronizations are Rare. In most software, synchronizations should constitute a small portion of all operations—most methods' invocations and exits, and most heap accesses will not cause threads to block or wake up. Further more, within any acquire/release window as illustrated in Figure 2, it is unlikely for the same synchronization to occur for many times — a non-synchronization like reading a popular variable or the invocation of a popular API could occur for many times, but a synchronization typically does not.

This hypothesis well complements the mostly-protected hypothesis, as the latter can be easily, yet incorrectly, satisfied by identifying all operations that ever appear in an acquire/release window as synchronizations.

Hypothesis: Acquisition-Time Mostly Varies. Intuitively, how long a thread needs to wait during an acquire varies a lot at run time. For example, when a thread acquires a lock, it could get the lock immediately if no one else holds the lock or take a long time if many threads are competing for the lock. Specifically, SherLock applies this hypothesis to every method: if every execution of a method m takes roughly the same amount of time, the invocation of m is unlikely to be an acquire.

This hypothesis directly helps identify methods that are used for acquire synchronization. In theory, it could also help identify acquire synchronizations implemented as code structures surrounding variable reads (e.g., while-loops). However, since SherLock does not attempt to identify such code structures, this hypothesis does not directly help identify variables used for acquire synchronization.

Hypothesis: Mostly Paired. Given the strong semantic connection between a release and its corresponding acquire, they are often defined in a paired or clustered way in well-maintained software. Specifically, if the read of a variable `C::v` is used for acquire synchronization, the corresponding release synchronization is very likely the write of the same variable `C::v`. As for methods, if a method of a class C is involved for release synchronization, its corresponding acquire is often a method that belongs to the same class C. For example, in C#, invoking `Monitor.enter` is an acquire, and its corresponding release synchronization is the exit of method `Monitor.exit` from the same system class `Monitor`. This hypothesis helps identify a release, once its corresponding acquire is identified with high confidence, vice versa.

Note that, we intentionally do not *require* an acquire operation and its corresponding release operation to come from the same class as this may sometimes not hold in practice. For example, when forking a thread, the release operation is `Thread::Start()`, from the C# system class `Thread`, but the corresponding acquire operation is the thread delegate from a class defined by developers. Yet, the `Thread` class does contain a release operation `Thread.Join()` matching our hypothesis.

3 HOW TO FACILITATE INTERESTING BEHAVIORS?

Challenges. Most of the hypotheses discussed in Section 2 are about dynamic software behavior. Given the non-determinism of

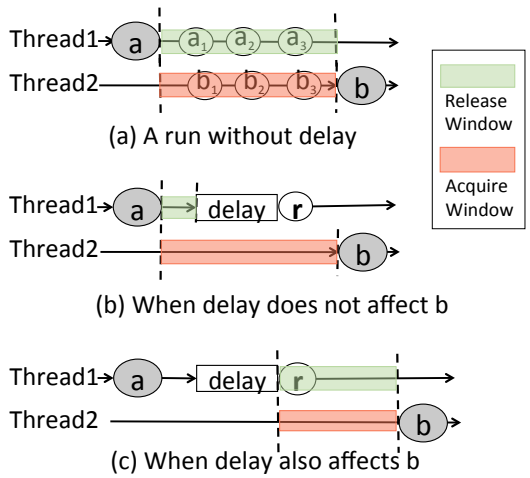

Figure 2: Acquire/release windows for synch. inference

concurrent software's dynamic behavior, we may not draw inference conclusions based on just one run.

For example, for any pair of conflicting accesses a and b, the physical time gap between them likely varies across runs and hence which operations appear in their acquire window and release window vary from run to run. When a and b happen to execute far away from each other, the acquire window and the release window will contain too many operations to be useful for synchronization inference. On the other hand, if a and b happen to execute close to each other, with few operations in between, such a behavior would be extremely useful and will be referred to as interesting behavior below.

However, a straw-man approach that simply re-executes the program many times is ineffective. Considering the huge interleaving space of a program, such a passive approach may never observe interesting behavior even after many runs.

To better handle this challenge, SherLock actively injects delays at strategic locations based on its inference from earlier runs to facilitate more interesting software behavior to occur.

Delay injection. At high level, in every run, SherLock injects delays around those top synchronization candidates based on previous runs, so that the software's reaction towards these delays can either strongly support or strongly dispute existing inference results, allowing true synchronizations to surface after a small number of runs (1–3 runs in our evaluation).

The exact strategy is illustrated in Figure 2. Imagine that SherLock has inferred r to be the most likely release synchronization between conflicting accesses a and b based on the observations collected so far, as shown in Figure 2 (a). In the next run, SherLock would inject a delay right before r.

This delay injection will produce valuable feedback no matter how the execution reacts. If, as shown in Figure 2 (b), this delay in the thread of a fails to cause a cascading delay in the thread of b, we can conclude that r is actually not the release coordinating a and b, and that the real release, if exists, should be between a and r — a much smaller release window than the initial window between in a and b in Figure 2 (a).

On the other hand, if, as shown in Figure 2 (c), the delay manages to propagate to the other thread, we can trust the current inference results more. Furthermore, we can confidently refine the acquire window to be between r and b, also a smaller window than the initial a—b window in Figure 2 (a).

Note that our delay injection and delay-propagation observation is similar to TSVD [28], which uses delay injection to identify thread-safety violations and uses delay-propagation to infer happens-before relation between conflicting thread-unsafe API calls. In contrast, the goal of delay injection in SherLock is to refine the acquire/release windows and delegates the actual inference of synchronizations and implied happens-before relations to the Sovler.

This is because delay injection can only help refine the inference process but cannot replace it. If there are too many release candidates, there will be too many delays injected, which not only takes long time but also makes it difficult to judge whether a delay has propagated. All the properties and hypotheses discussed in Section 2 help the Solver to identify a small number of highly likely release candidates. Without them, delay injection alone can hardly discover real release and acquire synchronizations. We will explain the exact implementation of SherLock Perturber in Section 4.3.

4 SHERLOCK

We now describe the design and implementation of various components of SherLock: the Observer, the Solver, and the Perturber. We also discuss how they work together.

4.1 Observer

Operations to trace. SherLock instruments a given application binary to trace two types of operations during run time.

First, it traces read/write operations that may form conflicting-access pairs useful for Mostly-Paired hypothesis. The operations include (1) read from or write to heap variables (e.g., public fields of a class), (2) getter and setter methods of public properties of a class, and (3) invocations of read/write APIs of thread-unsafe libraries (e.g., `List.Add()`). Note that, this API list is optional. In the current prototype, SherLock instruments 14 well-documented thread-unsafe C# library classes in the `System.Collections.Generic` namespace. Even without this API list, SherLock can still infer synchronization based on conflicting heap accesses. In fact, in our experiments, SherLock only misses about 3% of the inferred operations when this API-list is empty.

Second, SherLock traces potential synchronizations including accesses to heap variables and entry and exit of methods. For application methods, SherLock instruments entry and exit points of their implementations. For library or system API calls, SherLock instruments immediately before and after the call sites.

Log-entry content. At run time, SherLock records the following information for an operation: (1) timestamp, (2) thread ID, (3) operation type: read, write, method entry, or method exit; (4) field name and its memory address for each read/write operation, and (5) method name and parent object id for each method entry/exit operation.

Forming acquire/release windows. In theory, we could identify every pair of conflicting accesses from the execution log and report an acquire/release window like the one in Figure 2.a. However, that would form overwhelmingly large number of windows. Particularly, when two conflicting accesses are far away from each other, there are too many operations between them to serve as useful hints.

Consequently, SherLock uses a physical time window *Near* to filter out less useful pairs (1 second by default). For every pair of conflicting accesses a and b that are nearby (i.e., $(T_b - T_a) \leq Near$, assuming a is before b without losing generality), SherLock extracts all the operations in the log that executes between T_a and T_b as release candidates (if they are from the thread of a) or acquire candidates (if from the thread of b).

Note that a static code location may execute multiple times and hence form many acquire/release windows, particularly when it is inside a loop. To be not overwhelmed with very similar windows from the same pair of static locations, SherLock sets a upper bound (15) for the number of windows that one location pair can form. After reaching the upper bound of a location pair, SherLock ignores subsequent windows for it.

4.2 Solver

The goal of the solver is to infer likely synchronizations from the observations collected from many runs. The key idea is to treat this as a probabilistic inference problem [12, 31]. Intuitively, if operations often appear to separate a pair of conflicting operations at runtime, then the probability of these operations being synchronization is higher.

Next, we explain the details of our encoding, with every constraint and objective function term representing one property or hypothesis discussed in Section 2.

Variables. SherLock encodes every synchronization candidate as a random variable, whose assigned probability indicates the likelihood of this candidate being a synchronization.

For field reads, SherLock generates random variables $read(f)^{acq}$ and $read(f)^{rel}$ to respectively represent the probability the operation is an acquire or a release. Field writes are treated similarly. SherLock identifies the variables with the fully-qualified type of the field (i.e., `ClassName::FieldName`), and assumes that all dynamic instances behave the same. That is, if a field is used as a synchronization once, it is always used thus. This reflects how synchronization variables are typically used in practice. More importantly, doing so enables SherLock to easily generate multiple observations for the same variable, in one run or across multiple runs, greatly improving the efficacy of inference.

For every method invocation, SherLock defines two variables, $begin(m)^{rel}$ and $begin(m)^{acq}$. Similar representations are used for method exits, $end(m)^{rel}$ and $end(m)^{acq}$. As in field accesses, SherLock associates all dynamic instances of a method to a single variable identified by its fully qualified type (i.e., `ClassName::MethodName`). SherLock additionally assumes that the synchronization behavior remains the same independent of the polymorphic types or parameter values, assigning various forms to the same underlying variables.

Constraints. SherLock encodes *Read-Acquire* & *Write-Release* property discussed in Section 2 as linear constraints that should never be violated. This encoding is straightforward, as we simply need to assign corresponding variables to be 0 (i.e., there is no chance for them to fulfill a specific type of synchronization).

$$\begin{aligned} \text{for any field, } read(f)^{rel} = 0,\ write(f)^{acq} = 0 \\ \text{for any method, } begin(m)^{rel} = 0,\ end(m)^{acq} = 0 \end{aligned} \tag{1}$$

In addition, SherLock encodes the *Single Role* assumption that any library API l is only used to serve one type of synchronization, either acquire or release, encoded as $begin(l)^{rel} + end(l)^{acq} \leq 1$.

Objective function. The properties above are encoded as *hard* constraints that can never be violated. SherLock also uses many hypotheses, which are essentially *soft* constraints, in that they can possibly be violated, but we want such violations to be rare. To represent these soft constraints, the basic idea, as in previous work [12], is to use a relaxed constraint $C \leq \epsilon$ and instruct the solver to minimize an objective function that incorporates ϵ, instead of strictly requiring $C \leq 0$. Using this idea, SherLock encodes the hypotheses in Section 2.

Mostly protected. Given $a_1, a_2, \ldots, a_n$ in a release window w^{rel} and $b_1, b_2, \ldots, b_m$ in a corresponding acquire window w^{acq} (as in Figure 2), SherLock formulates the hypothesis that "there probably exists an release synchronization among $a_1, a_2, \ldots a_n$" and "there probably exists an acquire synchronization among $b_1, b_2, \ldots b_m$" as minimizing the following two terms:

$$\begin{aligned} rel(w) &= max(0, 1 - \sum_i a_i^{rel}), a_i^{rel} \in w^{rel} \\ acq(w) &= max(0, 1 - \sum_i b_i^{acq}), b_i^{acq} \in w^{acq} \end{aligned} \tag{2}$$

When at least one of a_1, a_2, ... and a_n is assigned a high probability of being an release, the top term would become 0; otherwise, it remains a positive number. The similar trend applies to the acquire probability assignment. Consequently, by minimizing the sum of all the *rel* and *acq* scores from all observed release and acquire windows, we can support the hypothesis that most conflicting accesses are protected. Note that, an operation o may have multiple dynamic instances in an acquire or a release window, but we always only subtract its corresponding probability variable once. Otherwise, the *rel* or *acq* term can be easily minimized without any variable having a close-to-1 synchronization-probability.

Synchronizations are rare. To encode the hypothesis that there are few synchronization operations in the program, SherLock simply adds a regularization term, which is the sum of all the variables, to the objective function. Minimizing it will minimize the number of synchronization operations.

$$reg(v_i) = v_i \tag{3}$$

To encode the hypothesis that a synchronization is typically not invoked frequently in any acquire/release window, SherLock adds the following penalty for a variable based on its average number of occurrence in every window that it appears in.

$$rare(v_i) = 0.1 * (average\ occurrence\ time\ of\ v_i) * v_i \tag{4}$$

We choose the coefficient to be 0.1 so that most $rare(v_i)$ is between 0 and 1, similar as the range of the regularization term above and the variation term below.

Acquisition time mostly varies. SherLock calculates the standard deviation and mean of every method m's duration. SherLock then adds the following term to the objective penalty function, which helps prioritize those methods that have high duration variation when identifying acquire synchronization.

$$var(m) = (1 - percentile(CV(duration(m)))) * begin(m)^{acq} \quad (5)$$

Here, coefficient of variation (i.e., standard deviation divided by mean) represents how much variation a method m's duration has. This term $var(m)$ compares m with all other methods. The higher the variation is, the less penalty, ranging from 0 to 1, we get when inferring m to be an acquire.

Mostly paired. To capture the hypothesis that release method often comes from the same class as its corresponding acquire method, SherLock introduces the following penalty for every class c that contains candidate operations, which is minimized to 0 when the number of acquire synchronizations in c equals the number of release synchronizations in c:

$$pair\_c(c) = \left| \sum_{op \in c} op^{acq} - \sum_{op \in c} op^{rel} \right| \quad (6)$$

To capture the hypothesis that if the read of a field f is used to acquire, the write of f is often used to release, vice versa, SherLock introduces the following penalty score:

$$pair\_f(f) = |read(f)^{acq} - write(f)^{rel}| \quad (7)$$

Overall objective function. Putting the above terms together, the overall penalty objective function is the following:

$$\sum_{w} (rel(w) + acq(w)) + \lambda \Big[\sum_{c} pair\_c(c) + \sum_{f} pair\_f(f) \sum_{v} reg(v) + \sum_{v} rare(v) + \sum_{m} var(m) \Big] \quad (8)$$

Here, λ is a trade-off knob. It determines the relative weight between Mostly-Protected hypothesis and all other hypotheses in our inference. By default, SherLock sets λ to be 0.2. We will evaluate different settings of λ in our evaluation section.

Solving & Result interpretation. SherLock uses a state-of-the-art linear solver [2] to find an assignment to all the variables that collectively satisfies all the constraints and minimize the penalty computed by the objective function.

Given all the variable assignment, we then check which acquire-probability variables and release-probability variables are assigned 1, and identify corresponding operations as acquire and release synchronization accordingly.

An important point to note here that these system of equations do not have a *trivial* solution — say one that makes every operation a non-synchronization, or one that makes every write a release and every read an acquire. This is because we require not only that at least one variable in the acquire (release) window to be an acquire (release), but also that the number of synchronizations should be minimized. This together prevents trivial solutions. This is an important property that allows our system of equations to have meaningful solutions without requiring bootstrapping with user annotations.

4.3 Perturber and Feedback across Runs

SherLock executes the target application multiple times (3 times per input in our evaluation). Across runs, observations are accumulated; new inferences are made; delays are injected accordingly, which then facilitate new observations.

To accumulate the observation across runs, SherLock does not throw away any constraints or objective function terms obtained from previous runs. Instead, it keeps (1) adding new variables and corresponding constraints, if new synchronization candidates show up, (2) adding new objective function terms for every newly observed release window and acquire window, and (3) updating existing objective function terms, like the average occurrence of a candidate operation, the co-efficient of variance of a method's duration, etc. Since the variables in our linear constraint system correspond to static names of methods and fields instead of their dynamic instances, the number of variables is guaranteed to be bounded and correlating information across runs is straightforward.

A special type of observation that gets accumulated is that, sometimes, SherLock could observe a *data race*. This occurs when SherLock observes a concurrent execution of two conflicting accesses with every operation in the acquire (release) window guaranteed to not be an acquire (release) synchronization. This can happen when either the acquire (release) window is empty or every operation in the window is a write (read) operation. When SherLock encounters such a data race between accesses a and b, SherLock remembers it and removes all the Mostly Protected penalty term associated with the acquire and release windows between a and b in all runs.

After every run, given the solver's updated inference results, SherLock Perturber injects a 100 milli-seconds delay right before every[1] dynamic instance of every operation that is currently considered as a release synchronization by the solver (i.e., no delay is injected for the first run). SherLock then checks whether the injected delay is propagated. Depending on that, the Perturber notifies the Observer to adjust the observed acquire window and release window accordingly, as illustrated in Figure 2 (b) and (c).

5 EVALUATION

5.1 Methodology

We implemented SherLock using Mono.Cecil [5] binary instrumentation framework, and evaluated SherLock on the latest versions (as of April 2020) of 8 open-source projects from Github. We run available unit tests of these projects for our dynamic monitoring. 7 of these projects (all but App-6 in Table 1) are from the benchmark suite set up by a recent C# concurrency-bug detection paper [28]. Only 2 projects from that suite were not evaluated here, because one closed its source code recently and one does not contain any multi-threaded unit tests. We also randomly picked one C# application, App-6 in Table 1, from the search results for "race condition" in Github. Table 1 shows the details of all these applications: they are all well maintained and reasonably popular based on the number of

[1] We also tried injecting the delay probabilistically, but did not see much difference in inference results.

Table 1: Applications in benchmarks

ID	Name	LoC	#Stars	#Tests
App-1	ApplicationInsights	67.5K	306	1193
App-2	DataTimeExtention	3.1K	335	219
App-3	FluentAssertion	78.1K	1886	3729
App-4	K8s-client	332.4K	395	139
App-5	Radical	95.9K	33	798
App-6	RestSharp	19.8K	7363	92
App-7	Stastd	2.3K	125	34
App-8	System.Linq.Dynamic	1.1K	399	7

Table 2: SherLock inferred results after 3 rounds. The sum in the parentheses is the unique synchronizations across applications.

ID	Syncs	Data Racy	Instr. Errors	Not Sync
App-1	46	10	2	7
App-2	6	0	0	0
App-3	8	0	2	0
App-4	20	0	1	0
App-5	14	2	0	2
App-6	14	0	0	2
App-7	19	4	0	0
App-8	6	0	0	1
Sum	133 (122)	16	5	12

stars on Github; their sizes range from around one thousand lines of code to more than three hundred thousand.

We run the benchmark suite on a Windows 10 laptop, with Intel(R) i7-8750 CPU, 16G Memory. Our evaluation answers the following key questions: (1) What synchronization operations can SherLock identify? (2) How helpful are these inferred synchronization operations in data-race detection? (3) What false positives and false negatives did SherLock incur? and (4) How do different components and parameter settings of SherLock affect the inference results?

5.2 Overall Results

Table 2 shows the results of running SherLock on our benchmarks. The table reports the results after 3 runs. Later sections will evaluate how the results vary with the number of runs. As the table shows, SherLock successfully identifies many real synchronizations. Of the 133 synchronizations identified, 122 are unique across the applications. Surprisingly, many of them are non-traditional synchronizations such as relying on finalizers to be called after the instruction that makes an object unreachable. We discuss more details and how the inferred synchronizations help data-race detection in Sec. 5.3 and 5.4.

As with other probabilistic inference techniques, SherLock makes 33 misclassifications shown in Table 2. 16 out of these 33 participate in true *data races* (forming 8 data race pairs). These include accesses that should be marked `volatile` to prevent memory consistency issues [7] as well as bugs due to missing synchronizations. For 5 cases, SherLock honed in on the right synchronization neighborhood but failed due to limitations of our current C# instrumentation infrastructure. The remaining 12 are instances where SherLock erroneously identified nonsynchronization operations as synchronizations. We discuss false positives and negatives in Section 5.5.

5.3 What Synchronizations Are Inferred?

SherLock identified 122 unique synchronizations for the applications in our benchmarks.[2] Of these, 51 are release synchronizations and 71 are acquire synchronizations. We classify these synchronizations into 19 system-API-based synchronizations, 12 variable-based synchronizations, and 91 application-method-based synchronizations.

5.3.1 System-API-Based Synchronization. This class includes methods in libraries that provide synchronization primitives for applications. This includes classic methods such as `Monitor::Enter` and `Monitor::Exit` and specialized methods related to to asynchronous processing like `DataflowBlock::Post` and Receive. An example of the latter is shown in Figure 3.A. Here `Post` is a release synchronization that happens before the entrance of an event handler and `Receive` is an acquire synchronization that happens after the exit of the event handler.

Furthermore, SherLock successfully inferred not only above 1-to-1 acquire-release synchronizations, but also some n-to-1 or n-to-n acquire-release synchronizations like `WaitHandle::WaitAll` in Table 8, which allows one code snippet to wait for many other code snippets.

If one would manually annotate synchronizations, these APIs are the simplest to do and this annotation effort can be amortized across multiple applications. Unfortunately, only a small percentage of synchronizations (19 out of 122) fall in this class. Moreover, 13 of these 19 API methods are used in only one application, validating a long tail of API-based synchronizations even in the small set of applications we study.

5.3.2 Variable-Based Synchronization. Four applications in benchmarks use variable-based synchronizations, contributing to 12 out of the 122 synchronizations. These include 4 while-loop synchronization and 8 if-checking based synchronization. For example, Figure 3.B shows a variable-based synchronization inferred from application App-4. Here, `endOfFile` is a flag indicating if the file writing has finished. Thread $T1$ sets it to be `true` after flushing the buffer to file; Thread $T2$ uses a while-loop to wait for the flushing to finish.

5.3.3 Application-Method-Based Synchronization. This is by far the largest class of synchronizations, contributing to 91 out of the 122 inferred ones. In these cases, the application relies on happens-before relation on a method return or a method entrance for synchronization.

Applications can use a variety of mechanisms to enforce happens-before ordering of these methods. First, such ordering can be guaranteed by the language semantics. For example, C# ensures that all static fields are properly initialized. This enforces a happens-before relationship between the return of the static constructor to any use

[2] Table 8 and Table 9 list the exact synchronizations SherLock inferred.

```
 //Example A @App-7
_block = CreateMessageParserBlock()
T1: _block.Post(Event e)
    _block.Receive()
T2: Messagehandler(Event e){...}

//Example B @App-4
volatile bool endOfFile = false;
T1: this.endOfFile = true;
T2: while (!this.endOfFile){/*wait*/}

//Example C @App-2
ConcurrentDictionary<T,T> dayCache;
T1: dayCache.GetOrAdd(year,delegate d1);
T2: dayCache.GetOrAdd(year,delegate d2);

//Example D @App-7
task = new Task(Action a1)
task.ContinueWith(Action a2)
T1: Action a1(){...}
T2: Action a2(){...}

//Example E @App-1
T1: void TestInitialize(){...}
T2: void BasicStartOperationWithActivity(){...}
```

Figure 3: Inferred synchronization examples

of the object. Similarly, C# ensures that finalizers on objects only run when the object is not reachable. Thus, the instruction that removes the last reference of an object happens before the beginning of the object's finalizer. SherLock inferred these relationships with no prior knowledge of the language semantics.

Method ordering can also be enforced by registering them as callbacks to system APIs. In Figure 3.C, two threads invoke `GetOrAdd` method on a concurrent dictionary. The delegate provided as a parameter to `GetOrAdd` executes when the specified key is not in the dictionary and is guaranteed to be atomic with respect to other delegates from concurrent calls to `GetOrAdd` on the same dictionary. This semantics guarantees a happens-before relation between the return of `d1` and the start of `d2` (or vice versa). Without understanding the involved semantics of the `GetOrAdd` method, SherLock identified that the starts and returns of the two delegates as synchronization.

Figure 3.D shows another such example. The program registers an action `a2` to continue with a predefined task `a1` using the `ContinueWith` API. This API guarantees `a2` to execute after `a1` independent of the threads executing these methods. Again, without understanding the semantics of `ContinueWith` mechanism in C#, SherLock infers the return of `a1` as a release synchronization and the start of `a2` as the corresponding acquire synchronization.

Some application frameworks enforce happens-before relations. As shown in Figure 3.E, `Microsoft.VisualStudio.TestTools.UnitTesting`, a popular testing framework, provides a static method, `TestInitialize`, to set up test environments. This method is guaranteed to execute before any unit test, like the `BasicStartOperationWithActivity` test function in App-1. Here, SherLock correctly infers the return of `TestInitialize` as a release synchronization, and the start of all executed test methods as acquire without knowing its semantics or analyzing any code inside the testing framework.

Table 3: SherLock vs. manual annotation in race detection (Only first data race reported in each run is counted).

	# True Data Races		# False Data Races	
ID	$\text{Manual}_{\text{dr}}$	$\text{SherLock}_{\text{dr}}$	$\text{Manual}_{\text{dr}}$	$\text{SherLock}_{\text{dr}}$
App-1	0	4	263	14
App-2	1	1	0	0
App-3	1	18	31	2
App-4	0	0	32	15
App-5	2	1	0	6
App-6	0	3	31	12
App-7	0	2	33	1
App-8	0	0	1	1
Sum	4	29	391	51

5.4 How Helpful Are Inferred Synchronizations?

Synchronizations are crucial in reasoning about concurrent programs. We quantitatively evaluate the benefit of synchronizations inferred by SherLock by using them in a dynamic data-race detector that mimics FastTrack [17, 18]. Since the original FastTrack algorithm was implemented for Java applications, we re-implemented FastTrack for C#[3].

We created two variants. The first, referred to as $\text{Manual}_{\text{dr}}$, is equipped with a list of manually identified synchronizations. For every Java synchronization tracked by FastTrack, we annotated corresponding C# synchronization API. We supported `volatile` variables, wait-notify synchronization, barriers, and happens-before relations from static initialization as reported in the paper [18] and from manual code inspection. The second, referred to as $\text{SherLock}_{\text{dr}}$, *only* uses the synchronizations inferred by SherLock. We compared the two versions when running all the unit tests of our benchmarks and manually inspecting error reports for true and false data races. Since the FastTrack algorithm is sound only till the first reported data race, we only count and inspect the first-data-race report in any bug-detection run. If the reported first-race is a false positive, a subsequent true race may get missed.

Table 3 shows that $\text{SherLock}_{\text{dr}}$ reports more true data races (29, compared to 4) and fewer false data races (51, compared to 391) than $\text{Manual}_{\text{dr}}$. Our manual inspection showed that all the false data races reported by $\text{Manual}_{\text{dr}}$ are due to missed synchronizations. For example, 323 out of the 391 false data races reported by $\text{Manual}_{\text{dr}}$

[3]We could not find a publicly available state-of-the-art C# data-race detector. For example, RaceTrack [44] does not handle static constructors or garbage collection, which FastTrack handles, or modern C# features like task-parallel-library.

Table 4: Breakdown of false positives/negatives.

	SherLock	$\text{SherLock}_{\text{dr}}$	
	#False Sync.	#Missed Sync.	#False Races
Instr. Errors	5	3	17
Double Roles	2	1	15
Dispose	5	4	11
Static Ctr.	4	2	3
Others	2	2	5
Total	17	12	51

Table 5: Inference with or without certain hypothesis

	# Inferred Sync. Ops.		
	#Correct	#Total	Precision
SherLock	122	155	79%
w/o Mostly are Protected	0	0	n/a
w/o Synchronizations are Rare	112	271	41%
w/o Acq-Time Varies	106	152	70%
w/o Mostly are Paired	101	158	64%
w/o Read-Acq & Write-Rel	100	152	66%
w/o Single Role	111	156	71%

are related to not handling the numerous ways of creating and executing tasks in C#, like those from `TaskFactory`, `ThreadPool`, etc. $\text{SherLock}_{\text{dr}}$ eliminates most of these false positives by its inference. Improving tools like FastTrack this way with automatically inferred synchronizations is a key motivation of this work.

$\text{SherLock}_{\text{dr}}$ reporting fewer false data races than $\text{Manual}_{\text{dr}}$ is natural as the former uses more happens-before relations than the latter. What is surprising is that $\text{SherLock}_{\text{dr}}$ reports more true data races than $\text{Manual}_{\text{dr}}$. On investigating further, we believe this is because FastTrack guarantees only hold till the first data race report. It continues to report subsequent data races, but only in a best-efforts manner. If the first data race is a false one due to missing synchronization, the reduced quality of subsequent reports can prevent it from detecting true data races. $\text{SherLock}_{\text{dr}}$ suffers less from this problem and thus reports more true data races.

$\text{SherLock}_{\text{dr}}$ still reports 51 false data races. This is due to SherLock's failure to infer 12 synchronizations (shown under #Missed Sync column in Table 4), which we discuss below.

5.5 What Caused False Positives/Negatives?

As shown in Table 2, SherLock inferred 33 incorrect synchronizations. Of these 16 arise from 8 data race pairs. On manual inspection, we found that 2 of these data races resulted in test assertion failures, implying that these data races are harmful.

Table 4 summarizes the remaining 17 false positives. Every false synchronization inferred corresponds to a true synchronization that SherLock missed. These misclassifications result in $\text{SherLock}_{\text{dr}}$ reporting false data races due to missed synchronizations, which are both shown in the table. We only report the false negatives identified during our manual inspection of data-race reports from $\text{Manual}_{\text{dr}}$ and $\text{SherLock}_{\text{dr}}$. There might be other synchronizations that we might have missed.

5 false positives resulted from errors in our instrumentation framework. Our instrumentation uses heuristics to identify and skip compiler generated and library code. The heuristics mistakenly skipped some application methods and did not expose them to SherLock. These resulted in 3 missed synchronizations and 17 false data-race reports. We report this category separately in Table 2 as this is an error in our implementation but not in our algorithm. We plan to rectify these errors in future versions.

2 false positives arise due to SherLock's Single-Role assumption that acquire and release cannot occur inside the same system method. In C#, there are a few APIs that violate that assumption. For example, `UpgradeToWriteLock` releases a reader lock and then acquires a writer lock all inside one API. This resulted in 1 missed synchronization and 15 false data-race reports. Future SherLock can try turning the Single-Role assumption into a soft constraint.

The remaining 11 false positives arose because of SherLock's inability to refine the acquire/release window effectively. These include failures to identify the acquire pair for object disposals (5), the release pair for static constructors (4), and other synchronization (2). For instance, dispose functions are often called during garbage collection which can execute at a much later time after the pairing release instruction that removes the last reference to the object. Since SherLock's delay injection does not control the garbage collection, it was not able to refine the windows.

Finally, there is another source of false negatives that are not reflected in our evaluation. Like all dynamic tools, SherLock can only observe executed code, and hence cannot identify un-executed synchronization operations. Fortunately, modern systems all contain many test cases and SherLock can use any concurrent tests.

5.6 More Detailed Results

How hypotheses helped. Table 5 shows how different hypotheses and synchronization properties have helped in reaching the SherLock inference results. All the numbers discussed in this sub-section and related tables and figures are about unique synchronizations inferred from 8 applications.

The Mostly-Protected hypothesis is the most crucial one. Without it, the Solver will simply decide that no synchronization exists in a program. Apart from it, the Synchronizations-are-Rare hypothesis is also crucial: without it, the precision of SherLock drops from 79% to merely 41%. It well complements the Most-Protected hypothesis to make sure that not too many operations are tagged as synchronization. The other hypotheses and properties are also helpful, as removing any one of them leads to fewer true synchronization identified.

Due to the randomness of concurrent execution, we observed one new correct synchronization when removing the single-role hypothesis and two new correct synchronizations when removing the mostly-paired rule. Overall, SherLock covers almost all the correct inference results.

How Perturber and multi-run feedback helped. Figure 4 evaluates several design decisions in our Perturber and feedback mechanism across runs. The worst performing schemes are when we do not

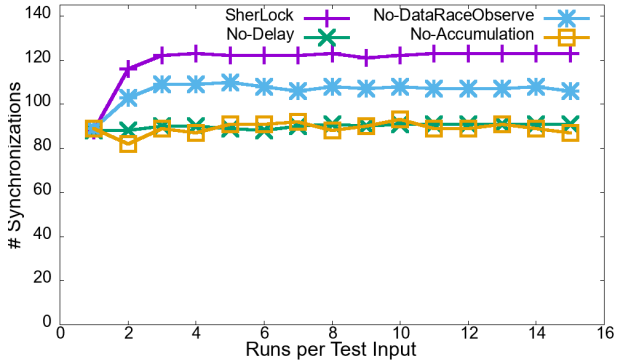

Figure 4: The numbers of correctly inferred unique synchronizations under different Perturber and feedback settings.

Table 6: Sensitivity of λ (numbers are the unique sums across 8 applications after running each test 3 times.)

λ	0.1	0.2	0.4	0.6	0.8	1	5	10	50	100
#correct	118	**122**	115	111	111	110	76	67	29	19
#total	157	**155**	156	147	144	142	95	85	36	29

Table 7: Sensitivity of `Near` (numbers are the unique sums across 8 applications after running each test 3 times.)

`Near`	0.01s	1s	100s
#correct	47	**122**	117
#total	85	155	183

accumulate constraints across runs at all (the yellow curve) and when we do not inject delays (the green curve). In both cases, the number of correctly inferred synchronizations drop from above 120 (SherLock) to around or lower than 90. The other decision that observes data races and removes corresponding Mostly-Protected terms from the objective function is also helpful (the blue line). We can also see that the number of correctly inferred synchronizations increases significantly for SherLock in the first three runs and becomes stable after that.

Parameter setting λ. Our overall objective function (Equation 8) contains a parameter λ that balances the weight of Mostly-Protected hypothesis term and all other terms. Table 6 shows the results under different settings of λ. When the λ increases, the weight of Mostly-Protected hypothesis decreases and hence infers fewer synchronizations; when decreases, SherLock infers more to ensure that every conflicting pair is protected. uses 0.2 as the default setting.

Parameter setting Near. We evaluated SherLock on three different settings of the physical-time window-size parameter *Near*, which was defined in Section 4.1. As shown in Table 7, when the window is too small (0.01 seconds), too few conflicting accesses were considered to be synchronized, causing SherLock to miss many synchronization operations. On the other hand, when the window is too large (100 seconds), the acquire/release windows contain too many synchronization candidates, which also caused SherLock to miss some correct synchronization operations, comparing with the 1-second default setting.

Enhancing TSVD inference. A recent thread-safety violation detector TSVD [28] infers happens-before relation between thread-unsafe API calls through delay injection and propagation check: when TSVD injects a delay at a thread-unsafe API call site a in thread 1 and observes this delay propagates to block thread 2, it infers that a happens before a thread-unsafe call site b in thread 2 right after the blocking period. This inferred happens-before knowledge is crucial for TSVD to save unnecessary effort in trying to expose thread-safety violation bugs among already synchronized calls. Note that, TSVD is not designed to pinpoint exact synchronizations and intentionally uses only simple heuristics to make quick inference, as low overhead is crucial for it. In this evaluation, we check if the synchronizations inferred by SherLock can help enhance TSVD happens-before inference.

We ran the open-source TSVD [4] for all applications in our benchmark suite. After 3 runs of every test input for every application, TSVD reports happens-before relation among 8 conflicting API-call pairs, with 7 of them truly synchronized (some applications do not call thread-unsafe APIs concurrently at all). By applying $\text{SherLock}_{\text{dr}}$ to the same set of benchmarks, 20 pairs of conflicting API-call pairs are identified as being truly synchronized—indeed, SherLock can be used to enhance TSVD in its happens-before inference and hence bug exposing.

Overhead. The overall overhead of applying SherLock to one test run, including instrumentation, tracing, and solving, ranges from 24% to 800%. The average overhead across all test cases is 278%, where tracing incurs 170% and solving incurs 94% overhead. In the default setting of running each test cases 3 times, the average overhead of using SherLock versus the baseline of executing the test cases 3 times without any instrumentation or delay, is 434%, where the delay injection introduces 156% overhead. Since we do not expect the current version of SherLock to be used in production, we consider the overhead of SherLock acceptable and have not worked on optimizing its various parts yet.

6 RELATED WORK

Happens-before identification and inference. Previous research worked on automatically identifying custom synchronization that uses synchronization variables. They [10, 40, 43, 45] apply static program analysis to identify specific program structures, like spin loop [40], shared-variable predicated control dependency[10, 43], and queues [45]. The applied static analysis is complicated and focuses on specific types of synchronization. In contrast, SherLock can identify various types of synchronization without sophisticated static analysis.

Some recent work infers the existence of happens-before relationship based on dynamic observation [11, 13, 28]. Mystery-Machine [13] infer task A happens-before task B if A always executes before B in millions of production runs. Orion [11] infers happens-before relation between two network operations whose time gap is nearly constant in thousands of runs. TSVD [28] infers a thread-unsafe API

Table 8: Inferred synchronizations from FluentAssertion, Radical and RestSharp

Synchronization	Description
App:FluentAssertion	
Release	
FluentAssertions.Execution.AssertionScope::.cctor-End	end of static constructor
System.Threading.Monitor::Exit	release lock
System.Threading.Tasks.Task::Run	create new task
Write-FluentAssertions.Specialized.ExecutionTime::<IsRunning>	write flag
Acquire	
System.Threading.Monitor::Enter	acquire lock
Read-FluentAssertions.Specialized.ExecutionTime::<IsRunning>	read flag
AssertionOptionsSpecs.When_concurrently_getting_equality_strategy.b2-Begin	start of task
FluentAssertions.Specialized.ExecutionTime::<.ctor>b0-Begin	start of task
App:Radical	
Release	
Microsoft.VisualStudio.TestTools.UnitTesting.Assert::IsTrue	end of last access
Radical.Model.Entity::EnsureNotDisposed	end of last access
Microsoft.VisualStudio.TestTools.UnitTesting.Assert::IsFalse	end of last access
System.Threading.Tasks.TaskFactory::StartNew	create new task
Radical.Messaging.MessageBroker::<SubscribeCore>-End	end of thread
System.Threading.Thread::Start	launch new thread
Radical.Messaging.MessageBrokerTests::<messageBroker_on_different_thread>	end of last access
Acquire	
Radical.ChangeTracking.ChangeTrackingService::Finalize-Begin	start of disposal
Radical.Model.Entity::Finalize-Begin	start of disposal
Radical.Tests.Model.Entity.EntityTests/TestMetadata::Dispose-Begin	start of disposal
Radical.Messaging.MessageBroker::<Broadcast>-Begin	start of thread
Radical.Tests.Windows.Messaging.MessageBrokerTests/TestRunner::<Execute>-Begin	start of thread
System.Threading.WaitHandle::WaitAll	wait for semaphore
Radical.Messaging.MessageBrokerTests::<broadcast_from_multiple_thread>_1	start of thread
Radical.Messaging.MessageBrokerTests::<broadcast_from_multiple_thread>_2	start of thread
App:RestSharp	
Release	
System.Threading.ThreadPool::QueueUserWorkItem	create new task
RestSharp.Tests.Shared.Fixtures.Handlers/<Generic>b30-End	end of task
System.Threading.EventWaitHandle::Set	release semaphore
RestSharp.Http::<WriteRequestBodyAsync>b2-End	end of task
System.Net.WebRequest::BeginGetResponse	send network request
System.IO.Stream::CopyTo	producer
RestSharp.RestClient<ExecuteAsync>b0-End	end of task
Acquire	
RestSharp.Http::<WriteRequestBodyAsync>b0-Begin	start of task
System.IO.Stream::Read	consumer
RestSharp.Tests.Shared.Fixtures.WebServer::<Run>b40-Begin	start of task
System.Threading.WaitHandle::WaitOne	wait for semaphore
RestSharp.Http::<WriteRequestBodyAsync>b0-Begin	start of message handler
RestSharp.Http::<GetStyleMethodInternalAsync>b0-Begin	start of event handler
RestSharp.Http::<WriteRequestBodyAsync>gRequestStreamCallback1-Begin	start of message callback
RestSharp.Tests.Shared.Fixtures.WebServer::<Run>b41-Begin	start of thread
RestSharp.Tests.Shared.Fixtures.TestHttpServer::<HandleRequests>b0-Begin	start of message handler

Table 9: Inferred synchronizations in DataTimeExtension, KubernetesClient and System.Linq.Dynamic.

Synchronization	Description
App:DataTimeExtension	
Release	
App.Common.ConcurrentLazyDictionary::GetOrAdd-End	end of atomic region
App.WorkingDays.EasterBasedHoliday/EasterCalculator::.cctor-End	end of static constructor
Write-App.WorkingDays.ChristianHolidays::ascension	write flag
Acquire	
Read-App.WorkingDays.ChristianHolidays::ascension	check flag
App.Common.ConcurrentLazyDictionary::GetOrAdd-Begin	start of atomic region
App.WorkingDays.EasterBasedHoliday/EasterCalculator::CalculateEasterDate	first access after static constructor
App: KubernetesClient	
Release	
Write-k8s.ByteBuffer::endOfFile	write flag: file is ready
System.Threading.Monitor::Exit	release a lock
k8s.Models.V1Status/V1StatusObjectViewConverter::ReadJson-End	end of await task
k8s.KubernetesClientConfiguration::GetKubernetesClientConfiguration-End	nd of await task
Write-k8s.KubernetesException::Status	write flag: meet error
Write-k8s.WatchLoop-End	end of await task
k8s.Yaml::LoadFromString-End	end of await task
k8s.KubernetesClientConfiguration::MergeKubeConfig-End	end of await task
k8s.KubernetesClientConfiguration::LoadKubeConfigAsync-End	end of await task
k8s.MuxedStream::Read-End	end of await task
Acquire	
System.Threading.Monitor::Enter	acquire a lock
System.Runtime.CompilerServices.TaskAwaiter::GetResult	wait for an await task
k8s.KubernetesClientConfiguration::MergeKubeConfig-Begin	await task beginning
k8s.StreamDemuxer::Dispose-Begin	await task beginning
k8s.ByteBuffer::Write-Begin	await task beginning
k8s.ByteBuffer::WriteEnd-Begin	await task beginning
Read-k8s.ByteBuffer::endOfFile	read flag: file is ready
k8s.ByteBuffer::Read-Begin	await task beginning
Read-k8s.KubernetesException::Status	read flag:meet error
k8s.KubernetesClientConfiguration::GetKubernetesClientConfiguration-Begin	await task beginning
App:System.Linq.Dynamic	
Release	
System.Threading.Tasks.TaskFactory::StartNew	create new Task
System.Linq.Dynamic.ClassFactory::.cctor-End	end of static constructor
System.Threading.ReaderWriterLock::DowngradeFromWriterLock	release lock
Acquire	
System.Linq.Dynamic.ClassFactory::GetDynamicClass	first access after static constructor
System.Linq.Dynamic.Test.DynamicExpressionTests::<CreateClass_TheadSafe>	start of thread
System.Threading.ReaderWriterLock::UpgradeToWriterLock	require lock

call *A* to happen before another thread-unsafe API call *B*, if an injected delay before *A* causes cascading delay to *B*. These techniques only infer the existence of synchronization between a pair of tasks or API calls, but cannot pinpoint the exact synchronizations that cause the effect. Instead, SherLock directly identifies the synchronizations. Furthermore, by considering multiple hypotheses and properties of synchronizations and feedback-based delay injection, SherLock does not require many runs to reach the inferring results.

Role inference in program analysis. Specification inference for program analysis is a well studied problem [12, 31, 36]. SUSI [36] trains a supervised support vector machine to identify the privacy roles in Android APIs. Merlin[31] and Seldon[12] use probabilistic

inference for identifying source, sink, and sanitizer specification for identifying security vulnerabilities.

SherLock is inspired by these works but applies them to the new setting of synchronization inference. This obviously requires new set of hypotheses and properties related to synchronizations, and leads to many differences in respective constraint systems: previous work [12, 31] uses non-linear constraints or linear constraints collected from many applications, while SherLock only uses linear constraints collected from the target application; SherLock updates its constraint system after every run, instead of statically [12, 31]. SherLock focuses on unsupervised inference while prior work requires to be bootstrapped with manually provided annotations.

Others. Decades of research has been conducted on analyzing concurrent programs, detecting concurrency bugs [9, 17, 20, 21, 25, 32, 34, 35, 38, 42, 46], and tuning performance of concurrent programs [3, 8, 13, 14, 29, 39]. SherLock is orthogonal to all these work and help them to achieve better analysis accuracy and capability with greatly decreased effort in annotating synchronization operations.

The hypothesis that mature software is mostly correct has also been used in statistical bug detection [15, 19], failure diagnosis [30], and inferring likely program invariants [16, 32]. SherLock shares similar philosophy with these work, but is solving fundamentally different problems and using completely different designs from them.

7 CONCLUSIONS

Synchronizations and happens-before relationship are fundamental in understanding and reasoning about concurrent programs. This paper made the first step in using unsupervised inference to identify synchronizations. The result shows that SherLock is effective in identifying various types of synchronizations using its well designed set of hypotheses and synchronization properties, assisted by its perturbation and feedback accumulation across runs.

ACKNOWLEDGMENTS

We would like to thank Brandon Lucia, our shepherd, and all reviewers for the insightful suggestion. This research is supported by NSF (grants CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039, CNS-1563956, IIS-1546543, CNS-1514256), the CERES Center for Unstoppable Computing, and the Marian and Stuart Rice Research Award. Dongjie Chen's research is supported by National Natural Science Foundation (Grants #61802165) of China.

A ARTIFACT APPENDIX

A.1 Abstract

SherLock is a tool that automatically infers synchronization operations that induce happens-before relationship in C# programs.

Its workflow contains three steps:

(1) Instrumenting the binary of the target application;
(2) Executing the instrumented binary with test inputs;
(3) Analyzing the log generated by testing runs and reporting identified synchronization operations.

SherLock can go through the above steps for multiple times using either the same set of inputs or different inputs, with its results from previous runs guiding following runs for better inference results.

We provide an artifact inside a virtual machine image, described in details below, to help easy reproduction of all the experiments described in the paper. In addition to that, SherLock source code is also available on GitHub at https://github.com/SpectrumLi/SherLock.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm**: Program instrumentation, log analysis, and linear constraint solving.
- **Binary**: A instrumentation tool written in C# and a log analysis tool in Python.
- **Run-time environment**: .Net 4.8 and Python 3.
- **Metrics**: The number of inferred synchronization operations.
- **Output**: A list of inferred synchronization operations for a target application, with the type, class name, method name, and line number specified for each such operation.
- **How much disk space required?**: The whole artifact, including virtual machine images, consumes about 50 GB of disk space. Directly downloading SherLock source code from the GitHub link above takes much less disk space, but would require more effort in setting up the experiment workflow.
- **How much time is needed to prepare workflow?**: The whole workflow to reproduce all the experiments in this paper is already prepared inside the virtual machine image — all it takes is to launch one script.
  To run your own experiments outside the provided virtual machine image, one can download SherLock source code from the link provided above and follow the set-up information described below.
- **How much time is needed to complete experiment?**: The total time needed to reproduce all our experimental results depends on the setting of your virtual machine. You should assign as many cores for the virtual machine as you can, if you want to finish the experiment quickly. In our in-house setting (Intel i7-9700k processor with 16 GB of physical memory), everything can finish within 2 hours.
- **Archived (provide DOI)**?: 10.5281/zenodo.4540866

A.3 Description

A.3.1 How to Access. The artifact is available in the zenodo website: https://doi.org/10.5281/zenodo.4540866

A.3.2 Hardware Dependencies. The tools and all the experiments should be run on a Windows machine.

A.3.3 Software Dependencies. SherLock requires the .NET framework to compile and instrument the target applications. It also requires a Flipy library for its python script to solve the linear constraints. These are all already installed and configured in the virtual machine image provided by us.

A.3.4 Data Sets. All the applications, including their test suites, mentioned in this paper are already pre-installed in the virtual machine image provided by us.

A.4 Installation

The provided Virtual Machine image has already installed everything needed for SherLock experiments. Here is a list of all the main files and directories in the Virtual Machine image:

- C:\TSVD\Benchmarks: This directory includes the source code of all the open-source applications used for evaluating SherLock. Note that, we only use AE-1, AE-2, AE-3, New-2, NAE-1, NAE-2, NAE-3 and NAE-4 directories, each representing one benchmark application. Other directories were generated when compiling these 8 applications. Inside each application directory, launch-prof-mem.ps1 is the script that you can use to compile, instrument, and run the application.
- C:\InstTool: This directory contains the SherLock instrumentation tool. It is a Visual Studio project, and we have already compiled it.
- C:\SherLock\idealy\log-analyze: This directory contains the SherLock solver tool, which takes execution logs as inputs and outputs inferred synchronization operations.
- C:\SherLock\Script: This directory contains the scripts that can launch the whole SherLock workflow and reproduce all the experimental results in this paper. The output files generated by running these scripts will also appear in this directory.

A.5 Experiment Workflow

Users can go to the directory C:\SherLock \Script and launch the following command inside our virtual machine:
.\Loop-delay-solve.ps1 [appname] [#round]

This command will instrument the specified application (`appname`) and run it for the specified number of times (`#round`), and output inferred synchronization operations into a local file described below.

That directory also contains a file runall-delay.ps1 that offers an example of calling Loop-delay-solve.ps1.

To apply SherLock on new applications, one can follow the example of scripts already provided in the artifact, or refer to the detailed Readme in SherLock GitHub repository.

A.6 Evaluation and Expected Result

The expected results are the inferred synchronization operations. In the prepared virtual machine, the output can be found in the directory C:\SherLock\Script named with result-[appname]-[#round].txt. This output file contains many debugging information and error messages generated by the original test suite, which you should all ignore. The real result of SherLock that you should read is at the end of this file in the following format:
Releasing sites:
Inferred-releasing-op-1
Inferred-releasing-op-2
Acquire sites:
Inferred-acquiring-op-1
Inferred-acquiring-op-2
...

REFERENCES

[1] Building async coordination primitives. https://devblogs.microsoft.com/pfxteam. Accessed: 2021-1-1.
[2] Flipy: linear solver. https://pypi.org/project/flipy/. Accessed: 2020-8-9.
[3] Ibm thread and monitor dump analyze for java. https://www.ibm.com/support/pages/ibm-thread-and-monitor-dump-analyzer-java-tmda. Accessed: 2020-8-9.
[4] Microsoft tsvd. https://github.com/microsoft/TSVD. Accessed: 2020-8-9.
[5] Mono.cecil. https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/. Accessed: 2020-8-9.
[6] Overview of synchronization primitives. https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives. Accessed: 2020-8-9.
[7] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. *ACM SIGARCH Computer Architecture News*, 18(2SI):2–14, 1990.
[8] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *EuroSys*, 2017.
[9] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.
[10] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, 2008.
[11] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, volume 8, pages 117–130, 2008.
[12] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 760–774, 2019.
[13] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 217–231, 2014.
[14] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.
[15] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.
[16] Michael Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.
[17] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.
[18] Cormac Flanagan and Stephen N Freund. The fasttrack2 race detector. Technical report, Technical report, Williams College, 2017.
[19] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.
[20] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. Asyncclock: Scalable inference of asynchronous event causality. In Yunji Chen, Olivier Temam, and John Carter, editors, *ASPLOS*, 2017.
[21] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race Detection for Event-Driven Mobile Applications. In *PLDI*, 2014.
[22] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.
[23] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.
[24] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.
[25] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.
[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
[27] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S Gunawi, and Shan Lu. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 994–1009, 2019.
[28] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.
[29] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.
[30] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
[31] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.
[32] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[33] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.
[34] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race Detection for Android Applications. In *PLDI*, 2014.
[35] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *PLDI*, 2012.
[36] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.
[37] Michiel Ronsse and Koenraad De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.
[38] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.
[39] Aater M. Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09*, pages 253–264.
[40] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.
[41] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *ASPLOS*, 2011.
[42] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-level detection of language-level data races with lard. In *ASPLOS*, 2014.
[43] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.
[44] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.
[45] Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, and Zhiqiang Ma. Atdetector: improving the accuracy of a commercial data race detector by identifying address transfer. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 206–215, 2011.
[46] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, 2011.