SherLock: Unsupervised Synchronization-Operation Inference
Extended Abstract

Guangpu Li¹, Dongjie Chen², Shan Lu¹, Madanlal Musuvathi³, Suman Nath³

¹University of Chicago, ²Nanjing University, ³Microsoft Research

1. Motivation
Tools for detecting [6, 11, 13, 14, 18, 24, 26, 27, 29, 32, 35] and fixing [15–17, 19, 25] concurrency bugs, understanding and tuning performance [2, 5, 9, 10, 21], and record-and-replay concurrent programs [28, 31] require understanding the synchronizations programs use. Typically, these tools rely on manual specifications of these synchronizations. Unfortunately, correctly identifying all synchronizations in modern software is challenging. Unlike textbook C programs, where a few pthread APIs perform all synchronization, modern software uses many different forms of concurrent execution such as traditional multi-threading, data parallel processing, event-based asynchronous computing, and others, and each form is coordinated by a varied set of synchronizations: C# standard threading library alone offers 5 lock and 9 signal-wait classes, with each containing many synchronization APIs and sub-classes [3]. Making things worse, programs can create their own, e.g., using shared variables, or use esoteric operating system facilities or even remote-server based synchronization. Incorrectly or incompletely identifying synchronizations can severely reduce the effectiveness of concurrency bug detection and performance analysis tools.

In this paper, we cast the synchronization inference problem as a dynamic unsupervised probabilistic inference problem. The basic idea is to dynamically monitor the operations during representative executions (say, during testing) for signals indicating synchronization behaviors. While each signal could be noisy, the goal is to cumulatively combine these signals over multiple executions to identify synchronizations. Being unsupervised has the advantage that one needs no user-provided annotations. This is essential for the general applicability of this technique.

2. Limitations of the State of the Art
Most work on analyzing concurrent programs relies on manually identifying synchronizations, which is tedious and error prone. This can lead to insurmountable porting effort for large modern software systems [20, 22].

Previous research has worked on automatically identifying custom synchronization but focused on specific program structures, like spin loop [30], shared-variable predicated control dependency [7, 33], and queues [34]. They typically require complicated static program analysis and only cover specific type of synchronization.

Some recent work automatically infers the existence of happens-before relationship between two tasks, either by observing consistent ordering between them [9] or by observing causal effects of delays around them [20]. Their goal is not to infer the exact synchronization used to enforce the happens-before relationship. Our work is thus orthogonal and can be used to improve the effectiveness of these tools.

This work is inspired by prior work on probabilistic inference for security specifications [8, 23], which identifies source, sink, and sanitizers for security-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.

3. Key Insights
Our unsupervised inference leverages three key insights.

Insight 1. Fundamentally, synchronizations order events that would otherwise result in bugs, with acquire synchronizations forcing a thread to wait (e.g., lock) and release synchronizations waking up a thread from its wait (e.g., unlock). For instance, a data race occurs when two threads concurrently access the same variable with at least one of them being a write, with no synchronization in between. Therefore, we hypothesize that most (if not all) such conflicting accesses in mature programs are properly synchronized. Thus, if one considers a...
Insight 1. While a single execution is insufficient to precisely identify which of the operations in the releasing window that follows \( a \) is a release synchronization and one of the operations in the acquiring window that precedes \( b \) is an acquiring synchronization.

We applied SherLock on 8 C# open-source applications. In total, by running each test input 3 times, SherLock automatically inferred 122 unique true synchronizations with few false positives. These include 1) standard synchronization primitives, such as monitors (\texttt{Monitor.Enter/Exit}), fork-join (\texttt{Task.Start/Wait}), and asynchronous tasks (\texttt{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 instructions and the object dispose). A version of FastTrack [11, 12] that we built for C# applications detects 7× more true data races and 8× fewer false data races by using these inferred synchronization information than the default.

This paper makes these contributions: (1) Identifying a set of properties and hypotheses reflecting fundamental assumptions about and usage of synchronization, that work together to enable effective synchronization inference. (2) A feedback-based delay injection scheme to actively expose run-time behaviors that help synchronization inference. (3) An artifact SherLock that uses unsupervised inference to automatically identify synchronizations with high coverage and accuracy.

### 6. Why ASPLOS

Synchronizations are critical to the correctness and performance of concurrent software. Unsupervised inference of these synchronizations will lead to more effective data-race detectors, as shown in this paper. We believe the techniques presented in this paper will power future bug-finding and performance-profiling tools, a topic that spans architecture, programming languages, and operating systems.

We strongly believe that future language runtimes, with appropriate hardware support, will infer nontrivial properties about programs and use them to improve program performance and correctness. We believe such inference has to be unsupervised to require no user annotations and dynamic to effectively use runtime program behavior. Looking back, one can consider branch predictors as unsupervised dynamic inference engines that are extremely effective. Inferring synchronizations is but a first step towards the vision above. Many problems left open in this paper, such as reducing the inference overhead (say using appropriate architectural support [4]), would be interesting for the ASPLOS community.
References

[10] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continu-

 processor tracing.

[21] Wei Zhang, Junghie Lim, Ramya Olichandran, Joel Scherpelz, Guo-