SherLock: Unsupervised Synchronization-Operation Inference
Extended Abstract

Guangpu Li[1], Dongjie Chen[2], Shan Lu[1], Madanlal Musuvathi[3], Suman Nath[3]

[1]University of Chicago,[2]Nanjing University, [3]Microsoft 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 subclasses [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.

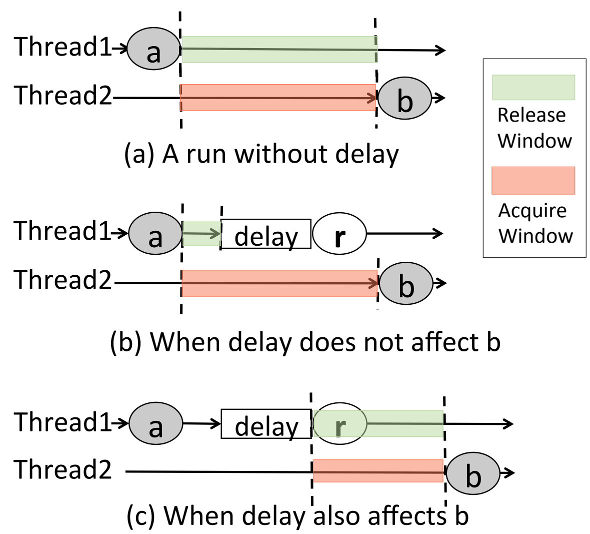

Figure 1: Identify synchronization in acquire/release windows

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

Variable reads cannot release Variable writes cannot acquire
Mostly, a pair of conflicting accesses are synchronized Mostly, an acquire op's duration varies much at run time Mostly, a synchronization is not invoked frequently
Most operations are for computation, not synchronization Mostly, if writing *v* helps release, reading *v* helps acquire

Table 1: Properties and hypotheses used by SherLock. (Gray-background hypotheses apply to dynamic behavior.)

dynamic execution shown in Figure 1.a with two potentially conflicting operations `a` and `b`, it is highly likely that one 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.

Insight 2. While a single execution is insufficient to precisely identify which of the operations in the releasing (acquiring) window offers a release (acquire) synchronization, we can do so by observing multiple executions and considering other natures of synchronization. Specifically, we design a set of properties and hypotheses (Table 1) that reflect fundamental assumptions of synchronization and their behavior.

Insight 3. Effective inference depends on conflicting operations, such as the ones in Figure 1, being temporally close. Otherwise, large acquire/release windows will produce too many synchronization candidates. Rather than relying on luck, we can actively perturb the execution at strategic locations to facilitate software behavior that is useful for inference and allow us to effectively identify synchronizations in just a few runs. As shown in Figure 1, imagine that prior observations indicate that `r` is a likely release synchronization. We can then inject a delay right before *r*. Next, if we observe that the execution of *b* does not get delayed together with *r* (Figure 1.b), we will obtain a much refined release window, between *a* and the delay, which squeezes out many incorrect release candidates, including *r*; on the other hand, if we observe that the execution of *b* also gets delayed (Figure 1.c), we will get a smaller acquire/release window, from *r* to *b* including *r*.

4. Main Artifacts

Guided by these key insights, we have designed SherLock. Given a program binary and test inputs, SherLock dynamically monitors the program on these tests. SherLock performs unsupervised inference on these observations using three components —an observer, a solver, and a perturber.

The Observer instruments the program binary to collect observations from every run and generates constraints on the likelihood a certain operation is an acquire or release using the hypotheses in Table 1. Some of the constraints are *hard* and cannot be violated (such as a read cannot be a release synchronization) while others are *soft* and thus can be violated, though we would like to minimize such violations.

The Solver encodes these constraints into a set of linear programming problem [8] and uses a linear solver [1] to identify candidate synchronizations.

The Perturber injects delays at strategic locations based on the Solver outputs to help the Observer gets more interesting observations and to help the Solver refines its results over runs.

5. Key Results and Contributions

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 (`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 [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

[1] Flipy: linear solver. https://pypi.org/project/flipy/. Accessed: 2020-8-9.

[2] 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.

[3] Overview of synchronization primitives. https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives. Accessed: 2020-8-9.

[4] Processor tracing. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[5] 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.

[6] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.

[12] Cormac Flanagan and Stephen N Freund. The fasttrack2 race detector. Technical report, Technical report, Williams College, 2017.

[13] 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.

[14] 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.

[15] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[16] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, 2012.

[17] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[18] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[19] 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.

[20] 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.

[21] 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.

[22] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[23] 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.

[24] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[25] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[26] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race Detection for Android Applications. In *PLDI*, 2014.

[27] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *PLDI*, 2012.

[28] Michiel Ronsse and Koenraad De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[29] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[30] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.

[31] 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.

[32] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-level detection of language-level data races with lard. In *ASPLOS*, 2014.

[33] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[34] 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.

[35] 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.