BARRIERFINDER: Recognizing Ad Hoc Barriers

Tao Wang, Xiao Yu, Zhengyi Qiu, Guoliang Jin, Frank Mueller
Computer Science Department
North Carolina State University
Raleigh, USA
{twang15, xyu10, zqiu2, guoliang_jin, fmuelle}@ncsu.edu

***Abstract*—Ad hoc synchronizations are pervasive in multi-threaded programs. Due to their diversity and complexity, understanding the enforced synchronization relationships of ad hoc synchronizations is challenging but crucial to multi-threaded program development and maintenance. Existing techniques can partially detect primitive ad hoc synchronizations, but they cannot recognize complete implementations or infer the enforced synchronization relationships.**

In this paper, we propose a framework to automatically identify complex ad hoc synchronizations in full and infer their synchronization relationships for barriers. We instantiate the framework with a tool called BARRIERFINDER, which features various techniques, including program slicing and bounded symbolic execution, to efficiently explore interleaving space of ad hoc synchronizations within multi-threaded programs for their traces. BARRIERFINDER then uses these traces to recognize ad hoc barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in recognizing ad hoc barriers automatically.

***Index Terms*—ad hoc synchronization; barrier; symbolic execution; interprocedural program slicing; Cloud9; LLVM**

I. INTRODUCTION

A. Motivation

A recent study [1] finds that programmers frequently implement their own *ad hoc synchronizations* for different reasons. Researchers were able to find 6 to 83 ad hoc synchronizations, such as ad hoc barriers, in each of the 12 studied program suites [1]. Ad hoc synchronizations are typically unmodularized and are difficult to understand and maintain. Fig. 1 shows an example to illustrate the basic concepts of ad hoc synchronizations. The ad hoc synchronization in Fig. 1 is formed by S_2 and S_3, where the shared variable `flag` is called a *sync variable*, the **`while`** loop in S_3 is a *sync loop*, S_2 is a *sync write*, and the sync loop and sync write compose a *sync pair*. The sync pair formed by S_2 and S_3 intends to enforce an order relationship that S_1 happens before S_4.

In general, because of the critical role that synchronizations play in multi-threaded programs, it is important to have an accurate understanding of their enforced synchronization relationships. The state-of-art techniques can only detect sync pairs [1] or simple busy-wait loops [2], [3], which is insufficient to understand complex ad hoc synchronizations. For instance, Fig. 2 shows another ad hoc synchronization with

This work was funded in part by the following grants: Air Force Office of Scientific Research AFOSR-FA9550-12-1-0442 and AFOSR-FA9550-17-1-0205, NSF 1217748, DOE 1403482, Lawrence Livermore National Laboratory subcontracts LLNL-B627261 and LLNL-B631308.

```
//Thread 1                          //Thread 2

counter = 5;  //S1                  while (flag); //S3
flag = false; //S2                  counter++;    //S4
```

Fig. 1: An ad hoc synchronization example formed by S_2 and S_3. `counter` and `flag` are global variables. `flag` is initialized to `true`.

the sync pair in lines 23 and 28 labeled, but it implements a barrier. During software maintenance, programmers may still experience difficulties in understanding the intended order relationship by the sync pairs and verifying their correctness [4]. Other multi-threaded program analysis tools, such as data-race detectors [5], concurrency-bug finding tools [6], automated bug-fixing tools [7] cannot simply depend on sync pairs either. Instead, they may also need other contexts, such as participating threads and the entire scope of synchronization constructs.

Therefore, detecting the entire scope of sync pairs and inferring their synchronization relationship is an important task but can also be very challenging. The ad hoc barrier in Fig. 2 exemplifies the two major challenges:

- First, a sync pair, which is the only information reported by existing ad hoc synchronization detectors, may be only a part of an ad hoc synchronization. Without considering extra code, it may be impossible to infer the enforced synchronization relationship. For example, the sync pair in Fig. 2, is only a portion of the complete ad hoc synchronization implementing a barrier. To recognize the ad hoc barrier, all the code from lines 12 to 29 needs to be considered, in addition to their threading context from lines 3 to line 6. In this example, the static control flow is already complex, and determining the threading context involves non-trivial interprocedural analysis.
- Second, there can be an excessive number of feasible thread interleavings to consider for inferring synchronization relationships and verifying their correctness. Without a thorough exploration or a proof, one cannot be sure what synchronization relationship is enforced by a sync pair and relevant code constructs or if the implementation is correct.

B. Contribution

To tackle these challenges, we propose an ad hoc synchronization analysis framework to (1) automatically recognize

```
1  int gsense = 1, gcount = 0, P = ...; // input
2  main() {
3    for (i=1; i<P; i++)
4      pthread_create(SlaveStart, ...);
5    ...
6    SlaveStart();
7  }
8  SlaveStart() {
9    ... // computation and two barriers
10   for (...) {
11     ... // computation
12     { // barrier begin
13     int lsense = gsense;
14     while (1) {
15       int oldcount = gcount;
16       int newcount = oldcount + lsense;
17       // atomic compare exchange using assembly
18       int updatedcount = CmpXchg(&gcount,
19                                  oldcount, newcount);
20       if (updatedcount == oldcount) {
21         if ((newcount == P && lsense == 1)
22               or (newcount == 0 && lsense == -1)) {
23           gsense = -lsense;     // the sync write
24         }
25         break;
26       }
27     }
28     while (gsense == lsense) ;  // the sync loop
29     } // barrier end
30     ... // computation and one barrier
31   }
32   ... // computation and one barrier
33 }
```

Fig. 2: Extracted code from SPLASH2 LU

complex ad hoc synchronizations beyond simple sync pairs, and (2) efficiently infer the enforced synchronization relationships without repetitively examining equivalent interleavings. To the best of our knowledge, no existing technique has accounted for such complexity.

We currently instantiate the framework for automatic recognition of ad hoc barriers and present BARRIERFINDER. We choose to focus on ad hoc barriers because they are both common and beneficial to be recognized. The ad hoc synchronization study [1] reported that barriers are a common type of synchronizations with ad hoc implementations. Further, a recent work also shows that the recognition of barriers can reduce the complexity of many multi-threaded program analyses and improve many development tools [8].

Our approach capitalizes on the intuition that all ad hoc barriers enforce a temporal invariant among different thread interleavings. Specifically, the temporal invariant requires that no participating threads can proceed beyond a program point (blocking point) before the last participant has reached a specific program point (releasing point), so that in effect computation prior to the blocking point are finished in all threads before computation after the blocking point can be executed in any thread.

As shown in Fig. 3, BARRIERFINDER takes program source code and sync pairs detected by SyncFinder [1] as inputs, and it proceeds in four steps to determine whether each sync pair and any relevant code compose an ad hoc barrier:

1) As a sync pair is seldom a complete ad hoc synchronization itself, we first slice the program with sync pairs as the slicing criteria. This helps us identify program constructs that are also integral parts of ad hoc barriers.
2) We then analyze and instrument the program slices with auxiliary API calls, such as scheduling and tracing API calls. They are directives to examine the temporal invariant of the sliced program constructs by efficient interleaving enumeration.
3) We further symbolically execute the program to exhaustively enumerate nonequivalent interleavings and to generate traces representing these interleavings.
4) Finally, we mine the interleaving traces to find predefined temporal patterns and infer the synchronization relationship. Since BARRIERFINDER focuses on ad hoc barriers, we define patterns for barriers. BARRIERFINDER reports whether a sync pair is part of an ad hoc barrier. If that is the case, it reports the complete barrier implementation.

Overall, this paper makes the following contributions:

- We propose a framework to infer the synchronization relationship enforced by ad hoc synchronizations. To our knowledge, we are the first to analyze ad hoc synchronizations beyond recognizing sync pairs.
- We instantiate our framework for ad hoc barriers and implement BARRIERFINDER with several novel techniques to account for interleaving space blow-up and to boost its execution efficiency.
- We evaluate BARRIERFINDER on real-world programs. Results suggest that our approach is efficient and effective in recognizing ad hoc barriers as a whole synchronization construct automatically.

II. EXAMPLE AND OVERVIEW

Below, we first describe the real-world example in Fig. 2 with details, and we then use it to illustrate the major steps of BARRIERFINDER, discuss the complexity of interleaving enumeration in the symbolic execution step, and show how BARRIERFINDER reduces the complexity with different techniques and optimizations.

A. An Illustration of the Major Steps

The code shown in Fig. 2 is extracted from a real-world program, SPLASH2 LU [9]. Within the `main` function, the parent thread first creates `P-1` child threads to execute `SlaveStart` and then also executes `SlaveStart`. Within `SlaveStart`, a total of five ad hoc barriers are used. Two of the five barriers are before the **`for`** loop in line 10, two in the **`for`** loop, and one after the **`for`** loop. Fig. 2 shows the code for the first ad hoc barrier in the **`for`** loop. The remaining barriers have the same code and are omitted.

For the ad hoc barrier from lines 12 to 29 in Fig. 2, SyncFinder [1] can only report a sync pair with a *sync loop* in line 28 and a *sync write* in line 23, not knowing that they are parts of this barrier. BARRIERFINDER, as shown in Fig. 3, takes source code and sync pairs reported by SyncFinder as input, and it then uses slicing to find more program constructs related to synchronization. For the example in Fig. 2, we use the reported sync write and sync loop in lines 23 and 28 as

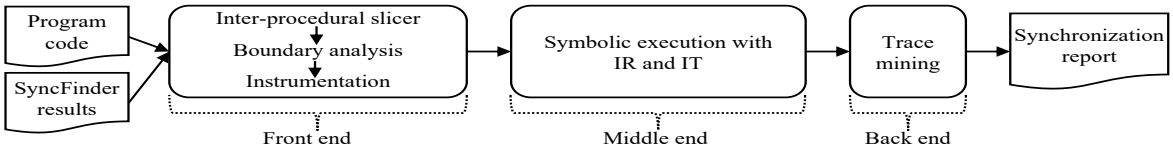

Fig. 3: The architecture of BARRIERFINDER. IR: interleaving reduction. IT: intrusive tracing.

the slicing criteria, and we are able to retain the entire code fragment from lines 13 to 28 after slicing.

To recognize ad hoc barriers in the sliced program, we rely on the temporal invariant exhibited by a barrier. We argue that all barriers exhibit the same temporal invariant, regardless of whether they are standard ones provided by languages/libraries or ad hoc ones. Specifically, if N threads in a program execute the barrier code, the first $N-1$ threads must always be blocked until the N-th thread unblocks them. As a result, if we collect a tracing event R immediately after the block operation in the $N-1$ threads and a tracing event W before the unblock operation in the N-th thread, then all traces of different interleavings must share the same pattern WR^N, i.e., a W (write) followed by N instances of Rs (reads).

Based on the observation above, our approach at a high-level is to gather program execution traces and mine the characteristic temporal invariant to recognize ad hoc barriers. To gather execution traces, BARRIERFINDER analyzes and instruments the program with trace API calls that generate different outputs representing the execution of different operations. In Fig. 2, BARRIERFINDER instruments a trace API call immediately before the sync write and another one immediately after the sync loop, so that they are executed before the sync write or after the sync loop.

With trace API calls instrumented in the sliced program, BARRIERFINDER uses a symbolic execution engine to carefully schedule the program execution, so that the same interleavings do not repeat. We cannot simply run the sliced and instrumented program under a native environment to collect traces. That is because different executions of the sliced program without explicit scheduling control in a native environment may only encounter a limited number of unique interleavings of the sync regions, and any mined temporal invariant may just be false. To symbolically execute the sliced program of the LU code shown in Fig. 2, we make the input variable `P` that determines the number of threads symbolic, and we start from value 2 for `P` during symbolic execution.

To guide the symbolic execution engine to explore unique interleavings, BARRIERFINDER further instruments the sliced program with scheduling API calls. BARRIERFINDER's symbolic execution engine executes one thread at a time and achieves concurrency by context switching among threads. The scheduling directive forces the symbolic execution engine to explore different interleavings by scheduling different threads. BARRIERFINDER only adds scheduling API calls after instructions that access shared variables, since they are the only interaction points among different threads. In Fig. 2, we have three shared variables, `gsense`, `gcount`, and `P`, with three, four, and one access(es), respectively. In particular, three of the four accesses to `gcount` are within the `CmpXchg` function in line 18. After instrumenting the trace and scheduling API calls, BARRIERFINDER collects traces corresponding to different interleavings and then checks traces against a predefined invariant representing barriers to recognize ad hoc barriers.

B. Complexity Reduction and Enumeration

Our approach requires an efficient enumeration of thread interleavings. For N concurrent threads each executing t instructions in a straight-line fashion, the combinatorial number of sequentially consistent interleavings is $\frac{(Nt)!}{(t!)^N}$. Such a large space presents a great challenge. Our solution entails both insights on this interleaving space and an ensemble of novel engineering techniques to achieve high efficiency.

Given the exponential interleaving space, we can first bound N and t to reduce the complexity. Nevertheless, exhaustive enumeration of all thread interleavings is still impractical. To make it feasible, we design a series of techniques to reduce the upper bound of the possible interleavings and optimize the enumeration process, so that our approach becomes feasible on complex multi-threaded programs. Next, we demonstrate these techniques on our example.

1) Scheduling Scope Reduction: We first introduce slicing-based *scheduling scope reduction*, which reduces the total number of instructions to be executed in the target program, and then heuristically partitions the instructions retained after slicing into *sync region*s. We consider a sync region as the unit that contains one high-level ad hoc synchronization.

During interleaving enumeration, our approach uses all the sliced sync regions as the scheduling scope instead of the entire program. As a result, the length of the program t in the complexity upper bound is reduced to the length of the sync region c, where c is significantly smaller than t in practice. If the number of threads in sync regions is sufficiently small, our analysis may be able to exhaustively enumerate all possible interleavings in a reasonable amount of time.

The LU code shown in Fig. 2 has five ad hoc barriers. One is fully shown in lines 13 to 28, and four others are omitted in comments. SyncFinder reports a sync pair for each individual barrier. After slicing with respect to these sync pairs, BARRIERFINDER can determine that the code from lines 13 to 28 is consecutive as it was before slicing, forming a natural boundary between synchronization and computation. As a result, our scheduling scope reduction technique uses lines 12 and 29 as the boundary to form the sync region for the barrier shown. Other barriers are handled similarly.

2) Avoiding Equivalent Interleavings: After scheduling scope reduction, there are still other types of enumeration

TABLE I: Overall results of BARRIERFINDER on SPLASH2 with slicing, intrusive tracing, and interleaving reduction

Benchmark	LOC.	LOB.	Slicing time	Patterns (#)	T	T_s	T_r	T_{st}	T_{sr}	T_{str}	$\frac{T_{sr}}{T_{str}}$
FFT	1.2k	4679	0.2 (0.001)	barriers (7)	OOR	OOR	57.6 (0.44)	OOR	17.4 (0.1)	1.3 (0.06)	13.4
Cholesky	6.1k	26479	94.8 (0.17)	barriers (4)	N/A	N/A	N/A	OOR	24 (0.3)	2.5 (0.06)	9.6
Raytrace	11k	24173	15.8 (0.04)	barriers (1)	N/A	N/A	N/A	8.6* (0.06)	17.4*(0.06)	8.3*(0.08)	2.1
Radix	1.2k	3856	0.1 (0.02)	barriers (7)	OOR	OOR	OOR	OOR	108.8 (1.0)	4.5 (0.17)	24.2
LU	1.1k	4555	0.53 (0.001)	barriers (5)	N/A	N/A	N/A	OOR	31.3 (0.2)	1.7 (0.01)	18.4
FMM	5k	16583	18.2 (0.1)	barriers (10)	OOR	OOR	355.4 (7.8)	OOR	333.5 (1.6)	12.3 (0.08)	27.1

inefficiencies due to interleaving equivalence, namely, interleavings which have the same execution context. Since a program's behavior depends only on its current states not its historical schedulings or states, equivalent interleavings are guaranteed to produce the same results in the future. To avoid enumerating equivalent interleavings, we use a context-based equivalence testing technique to reduce all equivalent interleavings.

For consecutive sync regions, such as the two barriers omitted in line 9 in Fig. 2, we perform the testing at the ending boundary of each region. Assuming r consecutive sync regions each with I interleavings, the complexity of naively enumerating all of them is $O(I^r)$ without equivalence testing. With equivalence testing, only one interleaving will continue its execution at the end of each sync region at best while all equivalent others are terminated, and the complexity of exploring all of them is reduced to $O(I * r)$. We call this technique interleaving reduction (IR).

3) Intrusive Tracing: At first, per-interleaving traces are initially generated and stored in a trace buffer within the address space of the program being interpreted. Whenever an interleaving is terminated, BARRIERFINDER symbolic execution engine directly accesses and dumps the trace buffer into a trace file without the interpretee's involvement for maximal execution efficiency. We call this technique intrusive tracing.

III. EXPERIMENTAL EVALUATION

A. Methodology and Experimental Settings

We implemented BARRIERFINDER's front end on top of LLVMSlicer [10], and the boundary analysis and instrumentation pass is implemented as a sub-pass inside the slicer. BARRIERFINDER's middle end is built on top of Cloud9 [11], for its flexible interpretation and symbolic execution capabilities. The back end is a stand-alone python package, which separates collected sync traces into independent ones and maps them back to their corresponding program source code contexts.

We conduct empirical experiments to evaluate the efficiency and effectiveness of BARRIERFINDER on the SPLASH2 [9] suite. All measurements are conducted on a machine with Intel Core i7-4790 @ 3.60 GHz (hyper-threading enabled), 16GB DDR3@1666 MHz memory, and Ubuntu 15.10 as the operating system. SPLASH2 is used in Xiong et al. [1]'s ad hoc synchronization study as a representative suite for scientific applications. These programs contain complex control flows and many functions in order to show the intellectual merits of BARRIERFINDER.

BARRIERFINDER takes sync pairs as input. Since SyncFinder [1] is no longer maintained by the original authors and the code is not available to us, sync-pair annotations are manually inserted. Note that sync pairs are low-level primitive synchronization constructs in that they are just busy-wait loops and write accesses to shared variables. They are neither complete implementations of ad hoc synchronizations nor do they indicate the enforced synchronization relationships.

B. Results on Real-World Benchmarks

Tab. I shows the results for the six SPLASH2 benchmarks currently supported by BARRIERFINDER. Column "LOC" lists the number of lines of C source code, and column "LOB" lists the number of lines of LLVM bitcode. We then show the slicing time of BARRIERFINDER's front end and the number of ad hoc barriers (column "Patterns (#)"). We next show the runtime of BARRIERFINDER to exhaustively enumerate the interleavings with the number of threads bounded to 2. For the remaining columns, subscripts r, s, and t represent interleaving reduction, program slicing, and intrusive tracing, respectively. Different subscript combinations show the runtimes consumed by BARRIERFINDER's middle end with different optimizations enabled. For example, T_{str} is the runtime with all three optimizations enabled, while T_{st} is the runtime with slicing and intrusive tracing enabled but interleaving reduction disabled. N/A indicates benchmark crashes, and OOR indicates the execution runs out of memory. Runtimes (in seconds) are averaged for 10 runs, with their standard deviations in parentheses.

We make the following observations from our results:

① BARRIERFINDER is effective in detecting different numbers of ad hoc barriers in these benchmarks, and we manually confirmed that BARRIERFINDER detects all the barriers in each benchmark. To the best of our knowledge, BARRIERFINDER is the first tool to have such a capability. No prior work, including SyncFinder [1], can detect any of these ad hoc barriers for what they are. Our predefined trace pattern for barriers is WR^N as mentioned in II-A. The trace generated for two consecutive barriers in LU is $11WRR22WRR$ with 11 and 22 as the trace separators. The two characteristic sub-traces WRR match our predefined temporal invariant, and their corresponding sync regions are accordingly reported as barriers. The sync regions contain both the upper and lower loops in Fig. 2. The detected pattern and sync region reports show that BARRIERFINDER is able to pinpoint the entire code construct of ad hoc barriers and recognize their barrier semantics. Note that it is possible for BARRIERFINDER to report false positives since it can only exhaustively enumerate the interleaving space when there are only two participating threads. However, there is no such case in our evaluation and we are unaware of such code in practice. False negatives cannot occur solely due to the bounded number of threads,

because an ad hoc barrier has to enforce the temporal invariant even if there are only two participating threads.

② BARRIERFINDER is efficient in recognizing ad hoc barriers. Specifically, column "T_{str}" in Tab. I shows the time spent in the middle end, which is usually less than 10 seconds for two threads. This shows our optimization techniques, combined together, make our approach quite efficient.

③ Interleaving reduction is the critical technique that enables BARRIERFINDER to efficiently enumerate the interleaving space of ad hoc barriers. Column "T_{st}" shows the runtimes of the middle end with slicing and intrusive tracing enabled but without interleaving reduction (IR). Except for Raytrace that contains only one barrier, all benchmarks run out of memory resources in less than two minutes and progress is very slowly after that. In comparison, runtimes in column "T_{str}" show that IR is critical for BARRIERFINDER's efficiency.

④ Slicing is critical for BARRIERFINDER's middle end to succeed in analyzing the benchmarks. As shown in column "T_r", without slicing, BARRIERFINDER's middle end crashes for Cholesky, Raytrace, and LU. The cause is rooted in Cloud9, but all benchmarks succeed with slicing enabled. The slicing overhead for FFT, Radix, and LU is small, but it is higher for Cholesky and Raytrace. The general trend is that larger benchmarks incur higher slicing overhead. The benefit of slicing is that it eliminates code that is irrelevant to synchronization explorations and improves middle-end efficiency, which is substantiated by comparing T_{sr} and T_r for FFT. Without slicing, the runtime for Radix is also prohibitively high as its computation exhausts main memory quickly.

⑤ Intrusive tracing boosts BARRIERFINDER's middle-end performance by up to 27X. Column "$\frac{T_{sr}}{T_{str}}$" in Tab. I indicates a significant speedup due to our trace optimization technique, which crosses the interpreter-interpretee boundary.

IV. RELATED WORK

Several empirical studies related to synchronizations have been performed. Xiong et al. [1] characterize ad hoc synchronizations of representative open-source applications and find they are pervasive. Existing techniques [1], [2] use either static or dynamic approaches to detect sync pairs. We proceed further to detect complete synchronizations and recognize enforced synchronization relationships.

V. CONCLUSION AND FUTURE WORK

This paper contributes BARRIERFINDER, a pipelined framework to automate the recognition of complex ad hoc synchronizations that realize barriers. The experimental evaluation shows that BARRIERFINDER is able to detect barriers in 6 SPLASH2 benchmarks efficiently. In the future, we plan to extend BARRIERFINDER to support benchmarks with loops and more threads, as they may present new efficiency challenges. In addition, we want to evaluate BARRIERFINDER on other larger modern scientific benchmarks with OpenMP pragmas to help programmers diagnose data races.

Our approach recognizes ad hoc barriers by mining execution traces for temporal invariants. Invariant mining is a technique pioneered by Daikon [12], and our approach shares many common elements with Daikon, e.g., generating concrete traces and mining traces for invariants. Similar to our work, researchers have also explored temporal invariant mining for different purposes. Beschastnikh et al. [13] propose techniques to mine temporal invariants based on partially ordered logs, and CSight [14] further uses temporal invariants to model concurrent systems. CloudSeer [15] uses temporal invariants to model the workflow of cloud systems and then uses the models for monitoring purposes. Instead, we focus on inferring the synchronization relationship of ad hoc synchronizations.

REFERENCES

[1] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad hoc synchronization considered harmful," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010.

[2] A. Jannesari and W. F. Tichy, "Library-independent data race detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2606–2616, 2014.

[3] X. Yuan, Z. Wang, C. Wu, P.-C. Yew, W. Wang, J. Li, and D. Xu, "Synchronization identification through on-the-fly test," in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 4–15.

[4] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 426–438.

[5] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid Program Analysis for Determinism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012.

[6] W. Zhang, C. Sun, and S. Lu, "Conmem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 179–192.

[7] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 221–236.

[8] M. Das, G. Southern, and J. Renau, "Section-based program analysis to reduce overhead of detecting unsynchronized thread communication," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, Jun. 2015.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995.

[10] M. Chalupa, M. Jonáš, J. Slaby, J. Strejček, and M. Vitovská, "Symbiotic 3: New slicer and error-witness generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.

[11] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 183–198.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, 2007.

[13] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, ser. SLAML '11. New York, NY, USA: ACM, 2011, pp. 3:1–3:10.

[14] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[15] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 489–502.