Empirical Software Engineering manuscript No.
(will be inserted by the editor)

BARRIERFINDER: Recognizing Ad Hoc Barriers

Tao Wang[1] · Xiao Yu[1] · Zhengyi Qiu[1] · Guoliang Jin[1] · Frank Mueller[1]

Received: date / Accepted: date

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. We instantiate the framework with a tool called BARRIERFINDER, which features various techniques, including program slicing and bounded symbolic execution, to efficiently explore the interleaving space of ad hoc synchronizations within multi-threaded programs and collect execution traces. BARRIERFINDER then uses these traces to characterize ad hoc synchronizations into different types with a focus on recognizing barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in doing this, and BARRIERFINDER is also helpful for programmers to understand the correctness of their implemented ad hoc synchronizations.

Keywords Ad Hoc Synchronizations, Barriers, Program Slicing, Symbolic Execution, Temporal Invariants

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.

[1] EB2,Campus Box 8206, 890 Oval Dr, Raleigh, NC 27606
[2] E-mail: {twang15, xyu10, zqiu2, guoliang_jin, fmuelle}@ncsu.edu

1 Introduction

1.1 Motivation

In the current multi-core era (Herb 2005; Sutter and Larus 2005), multi-threaded programming has become imperative to leverage the full power of modern CPUs. As multi-threaded programs share resources across threads, programmers rely on proper synchronizations to ensure program correctness and efficiency. While a common set of *standard synchronizations*, such as mutex and condition variable operations, are provided by different languages or libraries, a recent study (Xiong et al. 2010) finds that programmers frequently choose not to use these standard synchronizations but implement their own *ad hoc synchronizations* for functionality or performance reasons. Researchers were able to find 6 to 83 ad hoc synchronizations in each of the 12 studied program suites (Xiong et al. 2010).

Because of the critical role that synchronizations play in multi-threaded programs, it is important to have an accurate understanding of the semantics of synchronizations and their enforced synchronization relationships. While standard synchronizations are easy to recognize and understand, ad hoc synchronizations have unmodularized implementations and enforce diverse synchronization relationships, making synchronization understanding a challenging task.

Fig. 1 shows an example to illustrate the basic concepts of ad hoc synchronizations. The ad hoc synchronization in Fig. 1 is formed by statements 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*. In this illustrating example, the sync pair formed by S_2 and S_3 enforces an order relationship between S_1 and S_4 that S_1 happens before S_4.

```
//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`.

To detect ad hoc synchronizations, researchers have already proposed various techniques (Jannesari and Tichy 2010, 2014; Tian et al. 2008, 2009; Xiong et al. 2010; Yin 2013; Yuan et al. 2013). However, existing techniques only detect sync pairs, i.e., sync loops and their corresponding writes, but they do not further infer synchronization relationships being enforced. This is problematic, as a sync pair can implement a mutual-exclusion relationship or different types of order relationships. For example, Fig. 2 shows another ad hoc synchronization with the sync pair in lines 23 and 28 labeled, but it implements a barrier.

Not only do programmers have difficulties in understanding the intended order relationship by the sync pairs and verify their correctness (Gu et al. 2015), but also multi-threaded program development tools, such as data-race detectors (Bessey et al.

```
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

2010; Lee et al. 2012), concurrency-bug finding tools (Park et al. 2009; Zhang et al. 2010), automated bug-fixing tools (Jin et al. 2011, 2012), synchronization determinism runtime (Cui et al. 2013; Zhao et al. 2019), and synchronization-oriented performance profilers (Chen and Stenstrom 2012; Yu and Pradel 2016), cannot directly use the ad hoc synchronization detection results of these existing tools. For example, SyncFinder (Xiong et al. 2010), which is the state-of-the-art tool for detecting ad hoc synchronizations, can detect the sync pair in Fig. 1, but it does not determine the order relationship enforced by S_2 and S_3. As a result, race detectors need to conduct further analysis on top of SyncFinder results. Otherwise, they could conclude that S_1 and S_4 constitute a data race on the shared variable `counter`, resulting in a false positive. To determine the order relationship in Fig. 1, one could enumerate all possible interleavings and see the temporal invariant that S_1 will always happen before S_4. The order relationship in this particular case is not difficult to determine due to the simplicity of this example.

However, inferring the synchronization relationship after detecting sync pairs is not always as easy as the one in Fig. 1, and sometimes it can 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, which includes the sync write on line 23 and the sync loop on line 28, is only a portion of the complete ad hoc synchronization implementing a barrier. To recognize the ad hoc barrier, all the code from line 12 to 29 needs to be considered, in addition to their threading context from line 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. Although the example in Fig. 1 has a small interleaving space and the synchronization relationship can be inferred with ease, the example in Fig. 2 has a much larger interleaving space, and the complexity of which will be detailed in Section 3.2. 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.

To sum up, understanding ad hoc synchronizations in terms of their semantics, i.e., the synchronization relationships being enforced, and correctness is an important but challenging task that has not been addressed. Techniques to bridge the gap, anywhere between existing ad hoc synchronization detection tools and various multi-threaded program development tools, are in a great need to make the results from the former be more useful for the latter.

1.2 Contribution

To tackle these challenges and bridge the gap, we propose an ad hoc synchronization analysis framework to (1) automatically recognize complex ad hoc synchronizations beyond simple sync pairs, and (2) efficiently infer the enforced synchronization relationships by exploring the interleaving space without repetitively examining equivalent interleavings. To the best of our knowledge, no existing technique has accounted for such complexity in the context of analyzing ad hoc synchronizations.

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 (Xiong et al. 2010) 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 (Das et al. 2015).

Our approach capitalizes on the intuition that all ad hoc barriers enforce a temporal invariant among different thread interleavings. Specifically, the temporal invariant involves a *blocking point* and a *releasing point*, and no participating threads can proceed beyond the blocking point before the last participant has reached the releasing

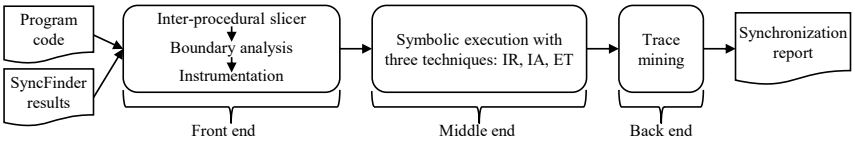


Fig. 3: The architecture of BARRIERFINDER. IR: interleaving reduction. IA: interleaving avoidance. ET: early termination. Trace mining: find pre-defined trace patterns.

point. As a result, computation prior to the blocking point shall be 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 in C as inputs, which will be annotated with sync pairs as detected by SyncFinder (Xiong et al. 2010). It proceeds in three major steps to determine whether each sync pair and any relevant code compose an ad hoc barrier:

1. To identify complete ad hoc synchronizations beyond sync pairs, we first perform program slicing for the target program with the annotated sync pairs as the slicing criteria. This helps us identify program constructs beyond sync pairs that are also integral parts of ad hoc barriers. We then analyze and instrument the program slices with auxiliary APIs, such as scheduling and tracing APIs. These APIs are directives to examine the temporal invariant of the sliced program constructs by efficient interleaving enumeration.
2. With the sliced-and-instrumented program LLVM bitcode, we then symbolically execute the program to exhaustively enumerate nonequivalent interleavings and to generate traces representing these interleavings. During symbolic execution, we bound the execution context and design several techniques to make exhaustive interleaving enumeration feasible and more efficient. We develop a runtime system that interprets the auxiliary APIs for efficient interleaving enumeration and trace generation.
3. 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. Otherwise, it reports the context of the violation.

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 both real-world programs and synthetic benchmarks. Results suggest that our approach is efficient and effective in recognizing different ad hoc synchronizations and can also help programmers understand the correctness of ad hoc synchronizations.

– We demonstrate how BARRIERFINDER's result can be further generalized to an unlimited number of concurrent threads with a proof sketch.

In our current prototype implementation and evaluation, the input C programs have a main thread that spawns several child threads. All threads participate in the same thread pool and carry out the same computation. We further assume that the barriers are counter based and every thread participates in them. Our evaluation shows that our prototype implementation is effective and efficient for programs and ad hoc barriers matching these assumptions. A short version of this work appears in the conference publication (Wang et al. 2019). This journal version contributes the following major extensions:

1. Section 3 provides more details on the BARRIERFINDER design that were only partially illustrated with an example in the conference version.
2. Techniques in Sections 3.2.3 and 3.2.4 are new, and they present novel solutions to solve challenges introduced by loops. Section 4.3.2 shows the evaluation results for the new techniques.
3. Section 4.4 is new, and it shows the effectiveness results of BARRIERFINDER on our synthetic benchmark suite and demonstrates how it can differentiate correct barriers from incorrect ones or non-barriers.
4. Section 4.5 is new, and it showcases a formal proof generalizing the characterization of counter-based ad hoc barriers in a program, e.g., the one in Fig. 2, to any number of participating threads in the program.

2 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 enumerating the interleavings in the symbolic execution step, and show how BARRIERFINDER reduces the complexity with different techniques and optimizations.

2.1 An Illustration of the Major Steps

The code shown in Fig. 2 is extracted from a real-world program, SPLASH2 LU (Woo et al. 1995). 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 (Xiong et al. 2010) 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 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 and after the sync loop, respectively.

With trace API calls instrumented in the sliced program, BARRIERFINDER uses a symbolic execution engine to carefully schedule the program execution, so that equivalent interleavings are not redundantly explored. 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 set the input variable `P` that determines the number of threads to 2. For `P` greater than 2, symbolic execution may not be able to exhaustively explore the interleaving space, and we provide an inductive proof in Section 4.5 with the `P=2` base as proved by BARRIERFINDER.

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 resembles a single-threaded machine and achieves concurrency of a multi-threaded program 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 program points where threads may interact with one another. For our example 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.

2.2 Complexity Analysis and Reduction

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.

2.2.1 Scheduling Scope Reduction

The *scheduling scope* is the subset of program source code where thread interleaving is enumerated. We introduce slicing-based *scheduling scope reduction*, which reduces the total number of instructions to be executed in the target program by excluding instructions that are not related to ad hoc synchronizations. We then heuristically partition the instructions retained with slicing into code sections, which will be referred to as *sync region*s. We consider a sync region as the basic program construct that may contain 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 of them is fully shown in lines 13 to 28, and four others are omitted in commented lines 9, 30, and 32. SyncFinder reports a sync pair for each of these five barriers. After slicing with respect to these sync pairs, lines 13 to 28 are retained after slicing. Since the program slice from lines 13 to 28 is consecutive with no holes, 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.

2.2.2 Avoiding Equivalent Interleavings

After scheduling scope reduction, there are still other types of enumeration 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).

For sync regions in loops, we can perform equivalence testing at the start of each region. In this case, if a sync region has the same execution context as one recorded in previous iterations, which we call a fixed-point interleaving, one can guarantee that the sync region will expose the same behavior as before and thus interleaving enumeration can be avoided. For the barriers in the `for` loop in Fig. 2, interleavings are only enumerated within a sync region during the first loop iteration to record new interleavings, and the equivalence test is performed before they are explored in later iterations. If found equivalent, interleaving enumeration will be avoided. This observation is leveraged by techniques interleaving avoidance (IA) and early termination (ET).

3 BARRIERFINDER Design

The key idea of BARRIERFINDER is to identify pre-defined trace patterns of ad hoc synchronizations by exhaustive enumeration of thread interleavings. Therefore, transformations employed by BARRIERFINDER, such as program slicing and instrumentation, must not change the target program's concurrency structure, i.e., cannot delete any existing synchronizations, insert new ones, or change the code on which any synchronization has a control dependency.

BARRIERFINDER employs various static and dynamic techniques to support efficient interleaving space enumeration. As shown in Fig. 3, there are three pipelined steps in BARRIERFINDER. The front end performs compile-time inter-procedural program slicing, sync region boundary analysis, and instrumentation on program LLVM IRs. The middle end symbolically executes the preprocessed program to collect sync traces with three critical techniques to tackle the challenge of efficient interleaving enumeration. BARRIERFINDER's back end analyzes sync traces and reports synchronization relationships for sync regions and their source code context information or the violation context. In this section, we elaborate on its design considerations and discuss alternatives when applicable.

3.1 Front-End Analysis and Instruction

3.1.1 Interprocedural Slicing

BARRIERFINDER employs program slicing (Weiser 1981, 1984) with the objectives to (1) reduce code size by elimination of irrelevant computations to achieve faster execution and (2) preserve the execution context for sync regions. To recognize ad hoc

barriers beyond sync pairs, such as those in Fig. 2, any viable approach has to capture the execution context of the sync region, including the initial values of sync variables, the number of participating threads, and sync region boundaries. BARRIERFINDER uses sync loop and sync write as the slicing criteria. Since the sliced program only provides interleavings allowed by the original program while minimizing the size of the sliced programs, we can mine the temporal property imposed by ad hoc synchronization via symbolically executing the sliced programs. Meanwhile, computation code is irrelevant in general for inferring the semantics of ad hoc synchronizations and may be sliced away to improve BARRIERFINDER's efficiency.

BARRIERFINDER's slicing step leverages LLVMSlicer (Slaby 2015), implementing Anderson's algorithm with field-sensitivity. We preserve program concurrency constructs in two ways. First, the slicer leaves all well-defined sync constructs (e.g, Pthread API calls) intact. Second, all potential accesses to sync variables are properly marked as sync writes and reads in sync loops based on SyncFinder results. Since sync loops and sync writes, which are the slicing criteria, correspond to reads and writes to shared memory locations that threads synchronize with, concurrent events generated by the original program and its sliced counterpart will be equivalent.

3.1.2 Sync Boundary Detection

Sync region boundary information marks the boundaries between computation and sync regions. Such boundary information is critical to generating separable traces for consecutive ad hoc synchronizations. Since there may be multiple ad hoc synchronizations in a program, e.g., the LU code in Fig. 2 has five barriers, BARRIERFINDER needs sync region boundary information to instrument *trace separator*s, which are characters used to distinguish consecutive synchronizations during the trace analysis stage.

A natural boundary is the first instruction sliced away with respect to the slicing criteria, and indicates the ending point of computation code prior to a sync region. To detect boundaries, BARRIERFINDER relies on a heuristic based on the observation that a sync region tends to have data dependencies only on sync variables, which are global or on the heap. BARRIERFINDER realizes this heuristic in a classical backward data flow analysis algorithm. This algorithm identifies (1) the first instruction accessing a sync variable as the start of a region and (2) the first instruction in the immediate post-dominator of a sync loop as the ending boundary. The algorithm in Fig. 4 utilizes classical backward data flow analysis. Gen/Kill sets are computed implicitly for each instruction traversed. For example, the load access of a global variable into a local variable would kill the local variable, and record the address of global variable as a live variable. In/Out sets are computed for the ending and beginning instruction of the analyzed basic block, respectively. The meet/join operator is the set union of the successors' Out sets. To account for the existence of loops, a fixed-point algorithm is applied. The algorithm terminates when the set of local variables is empty for the current instruction (line 14 in Fig. 4).

```
1  Input: CFG of the Function F where sync code resides,
2      worklist={basic blocks in the read-side loop}
3  Output: the boundary instruction, bi
4  Steps:
5  do{
6    do{
7      for bb in worklist
8        //reversely traversal of instructions in bb
9        for ins in bb{
10         //In[ins]/Out[ins] is In/Out set for ins
11         calculate In[ins], Out[ins];
12         if(worklist.size() == 1
13             && In[ins].localSet.size() == 0)
14           return ins;
15       }
16   }while(if any bb's In set changed);
17   /*compute new worklist for bb's whose successors have been visited.*/
18   for bb in worklist
19     push bb's unvisited predecessors and all its visited successors into
       newlist
20   for bb in newlist
21     compute bb's In Sets;
22   clear worklist;
23   copy newlist into worklist;
24 }while(worklist is not empty);
```

Fig. 4: Sync Boundary Detection Algorithm

3.1.3 Trace API Instrumentation

After detecting the boundary of a sync code region, BARRIERFINDER's front end instruments the region with trace API calls to collect traces during symbolic execution in the middle end. There are several considerations while deciding where to instrument trace API calls. First, we have to distinguish sync writes and sync loops by different tracing events. Second, we want to insert a minimal but sufficient number of trace API calls. Since trace generation is interpreted in BARRIERFINDER, it is necessary to minimize its performance overhead. Moreover, if too many runtime events are traced, temporal-invariant mining would suffer from unnecessary overhead. BARRIERFINDER relies on the following rules to satisfy these constraints:

- **Rule 1** Insert a trace API call that generates a character 'R' right after a sync loop.
- **Rule 2** Insert a trace API call that generates a character 'W' right before a sync write.
- **Rule 3** Insert a trace API call that generates a monotonically increasing separator as an integer counter at the beginning boundary of a sync region.

With **Rules 1** and **2**, one character that corresponds to one access to a sync variable will be generated, and these characters allow us to distinguish sync writes and sync loops. With **Rule 3**, we expect a separator to facilitate the distinction of traces over the detected ad hoc synchronizations. Note that BARRIERFINDER enumerates interleavings sync region by sync region. With these rules, we obtain traces like

11*WRR*22*WRR* for two consecutive barriers with 2 threads. It is then straightforward for BARRIERFINDER's trace analyzer to separate traces into several independent ones and to correlate them to sync regions.

3.1.4 Scheduling API Instrumentation

To guide the symbolic engine to enumerate different interleavings, BARRIERFINDER further instruments the sliced program with scheduling API calls. One may think of BARRIERFINDER's middle end as executing on a uni-core processor, i.e., there is only one running thread per execution state at any time, whereas concurrency among multiple threads is achieved via the instrumented scheduling API calls. BARRIERFINDER's middle end relies on scheduling API calls to know the timing for interleaving enumeration. A scheduling API call is a special function call instruction instrumented by BARRIERFINDER's front end.

To decide where to instrument scheduling API calls, we divide all instructions within a sync region into two categories.

1. Instructions accessing global variables or heap variables, which are visible to all threads, denoted as *IG*. *IG* can impose side-effects across threads and affect their execution.
2. Instructions other than *IG*, such as accesses to thread-local variables, denoted as *IL*. *IL* can neither "import" side-effects from other threads affecting its own execution nor "export" side-effects to affect other threads.

As different interleavings of instructions in *IL* do not change the global program state, it is sufficient to instrument a scheduling API after each instruction in *IG*. The scheduling API essentially instructs the symbolic execution engine to explore different scheduling decisions at each instrumented call.

3.2 Middle-End Symbolic Execution and Trace Generation

BARRIERFINDER's symbolic execution engine represents an interleaving with an execution state. During symbolic execution, the underlying engine *forks* new execution states while exploring different scheduling decisions, and there is a one-to-one mapping between execution states and interleavings. Calls to the scheduling APIs guide BARRIERFINDER's symbolic execution engine to explore new interleavings by forking new execution states. As shown in Fig. 5, there is only one execution state initially. When BARRIERFINDER's symbolic execution engine sees a scheduling API call, it forks new execution states for each possible interleaving. Suppose there are t threads, denoted as T_i ($1 \leq i \leq t$), ready to run in execution state ES_0, excluding the currently running thread T_0. When BARRIERFINDER interprets a scheduling API, it first makes t copies of ES_0, denoted as ES_i ($1 \leq i \leq t$). It then schedules T_i ($1 \leq i \leq t$) as the running thread for ES_i, as shown in the middle of Fig. 5. As a result, newly forked execution states ES_i ($1 \leq i \leq t$) and ES_0 only have different running threads. All other execution contexts and resources, i.e., registers, memory contents and data layouts, are exactly the same. Afterwards, each execution state is executed indepen-

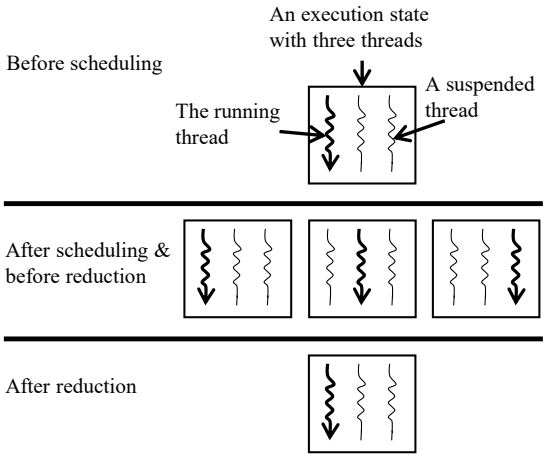


Fig. 5: Interleaving enumerations and reduction: An execution state in BARRIERFINDER corresponds to an interleaving.

dently to enumerate all interleavings within a sync region with repeatedly forked states at scheduling API calls until the program exits.

To tackle the challenge of interleaving space explosion, BARRIERFINDER's middle end relies on three techniques, namely interleaving reduction (IR), interleaving avoidance (IA), and early loop termination (ET). These techniques exploit interleaving equivalence to shrink the exponential interleaving space and enable BARRIERFINDER's efficient interleaving enumeration within sync regions.

3.2.1 Interleaving Equivalence Test – The Foundation

An interleaving at any point of execution corresponds to one execution state of a multi-threaded program. Two interleavings are different if they schedule different threads to execute at any of the same scheduling points. If the same set of sync traces can provably follow two interleavings, BARRIERFINDER considers them to be equivalent for the purpose of trace enumeration. Specifically, if BARRIERFINDER finds two interleavings with the same *execution context*, i.e., program pointers, calling stacks, and memory contents, across threads, it can guarantee their equivalence and reduce them to one without missing any distinctive future sync traces. This is because program execution only depends on the current execution context but not their past interleavings. However, this may be too restrictive for barriers, where thread identity does not matter. Taking this into consideration, BARRIERFINDER excludes identifiers of participating threads from the equivalence test. As BARRIERFINDER's middle end is a symbolic execution engine, where each interleaving of the sliced target program is executed by interpretation and its execution context is a managed data structure, the equivalence check is automatically performed by the middle end.

If two interleavings are equivalent when the interleaving equivalence test is invoked for a program point P at run time T, we call either one of them an interleaving

invariant with respect to the program point P at run time T. We refer to P as the invariant derivation point and T as the interleaving reduction time. The actual values of P and T are specified when the interleaving equivalence test is invoked, which are elaborated for interleaving reduction and interleaving avoidance in Section 3.2.2 and Section 3.2.3, respectively.

3.2.2 Interleaving Reduction – The Enabling Technique

Interleaving reduction (IR) reduces redundant exploration among equivalent interleavings. The program point P is a compile-time instruction in the target program, seen by BARRIERFINDER front end's static analysis; the reduction time point T to perform interleaving reduction is a run-time instruction seen by BARRIERFINDER's symbolic execution engine. They are identified with the following heuristics:

- P should be an instruction within a post-dominator basic block of sync region ending boundary that all participating threads execute. If only a subset of participating threads execute P, BARRIERFINDER might miss certain invariants.
- T should be within such a program execution state that it is likely for interleavings to be equivalent, e.g., when the first or the last participating thread passes P.

BARRIERFINDER selects the first instruction in a sync region's immediate post-dominator basic block as P, relative to the sync region's ending point. T is selected as the time when the last thread passes P. We implement a classical reference counter to maintain the number of threads entering and exiting a sync region. The counter is initialized to zero indicating that no thread is currently in the sync region. On entry, the counter is incremented; at the exit, it is decremented. When the counter is zero again, the runtime library knows that the last thread has just passed through the region such that its time T and the interleaving invariants can be derived.

Upon the first interleaving reaching T, there is no interleaving invariant yet. Hence, BARRIERFINDER adds this interleaving to the invariant set (*IS*), suspends its execution and schedules other interleavings for execution. For any subsequent interleaving reaching T, if an equivalent interleaving is found in *IS*, the new interleaving is terminated and all its resources are released immediately; otherwise, BARRIERFINDER adds it to *IS* as a new invariant. This process continues until all interleavings have been enumerated and executed. Then, BARRIERFINDER schedules and executes all invariant interleavings in *IS*. As illustrated in Fig. 5, 3 interleavings are assumed to be equivalent after they are forked and further explored independently. With interleaving reduction, they are reduced into one representative while the other two are terminated.

3.2.3 Interleaving Avoidance – Loop-centric Technique One

If a new interleaving to be explored has the same execution context as another interleaving that has already been explored in previous loop iterations, BARRIERFINDER can avoid exploring this new interleaving. Interleaving avoidance (IA) targets such opportunities for sync regions that are executed multiple times, especially those in loops. In contrast to interleaving reduction, interleaving avoidance (IA) is performed before a sync region is entered.

Specifically, BARRIERFINDER IA selects program point P as the first instruction in the immediate dominator basic block of a sync region's beginning boundary and time point T as when the first thread executes P. BARRIERFINDER IA adds the first interleaving hitting T to invariant set IS, snapshots its execution context, and continues its exploration as normal. When a subsequent interleaving hits T, BARRIERFINDER checks whether it is equivalent to any invariant in IS with the equivalence test. If successful, interleaving enumeration for this new interleaving is avoided. Otherwise, a new interleaving is identified that has not been explored before and the algorithm proceeds as normal. In this way, IA prevents redundant interleaving enumerations from being considered at all across loop iterations.

3.2.4 Early Loop Termination – Loop-centric Technique Two

We observe that there is usually left-over computation code after slicing. Such code is not sliced away due to limitations of interprocedural slicing. If such code is within loops, it may introduce a high execution overhead. Moreover, sync regions do not expose new traces after several loop iterations in practice. Early loop termination (ET) takes advantage of these observations by breaking out the innermost loop (ET-loop), which contains the sync regions being explored after the ET-loop *fixed-point state* is established. A *fixed-point state* for each sync region is established after a sync region has been explored under the same execution context twice. The *fixed-point state* for the entire loop is established when all encompassed sync regions have seen a fixed-point state at least once, which is also the point in time for ET to be applied. To terminate ET-loop early, ET sets the program counters of all threads to the first instruction in the immediate post-dominator basic block of the current ET-loop.

3.2.5 Optimizations for Execution Efficiency

We choose the state-of-the-art symbolic execution engine Cloud9 (Bucur et al. 2011) as the infrastructure for BARRIERFINDER's middle end. The execution engine is based on interpretation, which is slow compared to native execution. Therefore, it is critical to minimize the overhead related to interleaving enumeration.

BARRIERFINDER employs a snapshot mechanism to capture execution contexts, which are used in the interleaving equivalence test 3.2.1. To reduce the cost, BARRIERFINDER takes a minimal snapshot as follows. Let's suppose *SR* is a sync region being explored. For interleavings that are created when the first scheduling point within *SR* is hit at time c_0, their execution contexts (except for their thread state, i.e., suspended/running, etc.) are the same. As execution progresses, other execution contexts, e.g., global variables, may diverge. Therefore, it is sufficient to only snapshot those variables that are updated since c_0 and are still alive when the equivalence test is performed.

BARRIERFINDER's front end is responsible for liveness analysis and snapshot API instrumentation. Snapshotting is incrementally performed within the BARRIERFINDER middle end during sync region exploration. This minimizes a snapshot.

As an interpreter, BARRIERFINDER's symbolic execution engine interprets a target program (interpretee). The relationship between an interpreter and an interpretee

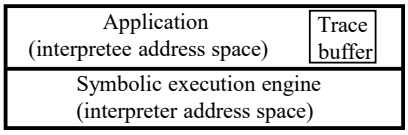


Fig. 6: Interpreter-interpretee address space layout in BARRIERFINDER: An application is interpreted in interpretee space while the interpreter itself uses native execution.

is similar to that between a CPU and process. The interpreter and interpretee have separate address spaces. The interpreter reads instructions from the interpretee's address space, evaluates it, and updates interpretee's execution state (data structures). Generally, the interpreter has limited knowledge on what the interpretee's internal execution state means and rarely performs actions relying on such knowledge. However, BARRIERFINDER does know where the interpretee's trace buffer is and capitalizes on such a fact. Fig. 6 shows the address space layout for BARRIERFINDER. BARRIERFINDER symbolic execution engine (interpreter) crosses the interpreter-interpretee address space boundary for native execution if possible. First, per-interleaving traces are initially generated and stored in a trace buffer within the interpretee's address space. BARRIERFINDER accesses and dumps its trace buffer into a file without the interpretee's involvement when an interleaving terminates. We call this technique intrusive tracing. Similarly, instead of relying on the interpretee to capture its execution context to a file, BARRIERFINDER snapshots a per-interleaving execution context in buffers within the interpretee's address space. BARRIERFINDER's middle end later reads these buffers directly, performs the equivalence test, and installs a new invariant in the interpreter's address space if necessary. All these happen in native mode without file I/O.

3.3 Back-End Trace Mining

BARRIERFINDER's back end analyzes sync traces to derive critical information for sync region understanding. Let's take a sample trace 11WRR as an example. First, BARRIERFINDER extracts the sync region ID 1 and use it to separate sync traces to two parts. It then checks whether the sub-trace is self-consistent, i.e., the number of 1s should be equal to the number of Rs since there is one trace point for each of them at the beginning and ending boundary of a sync region respectively. If self-consistency is confirmed, then BARRIERFINDER tries to match WRR with our pre-defined invariant for barriers WR^N. Since it succeeds for this trace example, BARRIERFINDER uses the extracted ID 1 as the key to retrieve context information, such as source code line numbers for the sync region and reports the code region as a barrier.

4 Experimental Evaluation

4.1 Prototype

We implemented BARRIERFINDER's front end on top of LLVMSlicer (Slaby 2015), 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 (Bucur et al. 2011), for its flexible interpretation and symbolic execution capabilities. Although BARRIERFINDER concretizes the thread number, the program slices can contain other symbolic variables. The back end is a stand-alone python package, which separates collected sync traces into independent ones, according to sync region IDs, and also maps them back to their corresponding program source code contexts. This way, BARRIERFINDER reports the number of recognized ad hoc barriers and their respective source code ranges.

4.2 Methodology and Experimental Settings

We conduct empirical experiments to evaluate the efficiency and effectiveness of BARRIERFINDER on the SPLASH2 (Woo et al. 1995) suite and a synthetic benchmark suite. All measurements are conducted on a machine with Intel Core i7-4790 @ 3.60 GHz (hyper-threading enabled), 16GB DDR3@1600 MHz main memory, and Ubuntu 15.10 as the operating system.

We use applications from the SPLASH2 suite to evaluate the efficiency of BARRIERFINDER, and we pay special attention to how different techniques speed up the interleaving enumeration process. SPLASH2 is used in (Xiong et al. 2010)'s ad hoc synchronization study as a representative suite for scientific applications. All the SPLASH2 programs match the assumptions made by BARRIERFINDER as we described in Sec. 1.2, i.e., they follow the thread-pool model and use a counter-based barrier implementation as shown in Fig. 2. Nevertheless, these programs contain complex control flows, including ad hoc barriers inside loops and consecutive loops, to show the intellectual merits of BARRIERFINDER.

To evaluate whether BARRIERFINDER can (1) handle counter-based barrier implementations that are different from the one in Fig. 2 and (2) correctly differentiate ad hoc synchronizations that implement barriers from those do not, we devise a synthetic benchmark suite to evaluate the effectiveness of BARRIERFINDER. Our synthetic benchmarks have a similar code structure but differ in ad hoc synchronizations. The main thread first creates one child thread, and all threads execute twice the same sequence of code, which is a variant of barrier based on the textbook "Synchronization Algorithms and Concurrent Programming" (Taubenfeld 2006). In total, we have eight different variants that are all counter-based (Malkis and Banerjee 2014; Taubenfeld 2006). Among the eight variants, two are correct ad hoc barriers, where one of them is the same as the ad hoc barrier in Fig. 2, and the other mainly differs in that it will reset `gcount` to `0` before line 23 and always increment `oldcount` by 1 in line 16. The other six are all wrong implementations because of different reasons, e.g., different initial values for sync variables, different orders of certain statements, and

whether or not atomic instructions are used. Because of these differences, these six wrong implementations violate the semantics of barriers or may result in deadlocks.

To show that our proposed framework is versatile, we also include one benchmark implementing an allAB (Jin et al. 2012) relationship extracted from MySQL in our synthetic benchmark suite. Specifically, the allAB relationship in our benchmark requires that the main thread executing the B operation cannot proceed to execute the B operation until both child threads cannot execute more A operations, and such an allAB relationship can be viewed as a variant of barriers.

Since these synthetic benchmarks stress less on interleaving-space enumeration, we focus on whether BARRIERFINDER can correctly recognize different ad hoc synchronizations and omit the performance measurements.

BARRIERFINDER takes sync pairs as input. Since SyncFinder (Xiong et al. 2010) 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.

4.3 Efficiency Results on Real-World Benchmarks

Tab. 1 shows the results for the six SPLASH2 benchmarks currently supported by BARRIERFINDER. Column "LOC./LOB." lists the number of lines of C source code and LLVM bitcode. We then show the slicing time of BARRIERFINDER's front end in column "Slicing time" and the number of ad hoc barriers in 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 s, t, and r represent slicing/boundary detection, intrusive tracing, and interleaving reduction, 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/boundary detection 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. To handle the default trip count of 32 interactions in LU requires interleaving avoidance and early loop termination, the results for LU in Tab. 1 are measured with trip count as 1 to show the benefits of slicing/boundary detection, intrusive tracing, and interleaving reduction on LU.

4.3.1 Observations

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 (Xiong et al.

Table 1: Overall results of BARRIERFINDER on SPLASH2 with slicing/boundary detection, 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

2010), can detect any of these ad hoc barriers in whole. The trace generated for two consecutive barriers in LU is 11WRR22WRR. BARRIERFINDER's back end divides such a string by considering 11 and 22 as separators. The two characteristic sub-traces WRR match our predefined temporal invariant for barriers, 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 detect the entire code construct of ad hoc barriers and recognize their barrier semantics.

It is possible for BARRIERFINDER to report false positives, i.e., BarrierFinder reports an ad hoc barrier but it is actually not one. False positives can happen since BARRIERFINDER can only exhaustively enumerate the interleaving space when there are a small number of participating threads, and there could be ad hoc synchronizations exhibiting different temporal invariants under different participating thread numbers. A false negative happens if BarrierFinder fails to recognize an ad hoc barrier or fails to characterize it correctly, which can happen if the sync write and sync loop are not identified in the first place. We assessed the quality of our evaluation results based on our understanding of the benchmarks, and we observed neither false negatives nor false positives in our evaluation.

② BARRIERFINDER is efficient in recognizing ad hoc barriers. Specifically, column "T_{str}" in Tab. 1 shows the time spent in the middle end, which is usually less than 10 seconds when there are two participating 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/boundary detection 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 very slowly after that. In comparison, runtimes in column "T_{str}" show that IR is critical for BARRIERFINDER's efficiency.

④ Slicing and boundary detection is critical for BARRIERFINDER's middle end to succeed in analyzing the benchmarks. As shown in column "T_r", without slicing/boundary detection, BARRIERFINDER's middle end crashes for Cholesky, Raytrace, and LU. The cause is rooted in Cloud9, but all benchmarks succeed with slicing/boundary detection 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/boundary detection 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/boundary detection, 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. 1 indicates a significant speedup due to our trace optimization technique, which crosses the interpreter-interpretee boundary.

4.3.2 The Benefits of IA and ET

In our current benchmarks, only LU, Radix, and FMM have loops that encompass sync regions. We manually adapt their loop trip counts to demonstrate the effectiveness of interleaving avoidance (IA) and early loop termination (ET). Specifically, we measure the runtime for the following configurations: (1) Interleaving reduction only (IR), (2) IR with IA but without ET (IR-IA), and (3) IR-IA with ET (IR-ET). Figs. 7a, 7b, and 7c show the performance results for Radix, FMM, and LU, respectively. The x-axis indicates the loop trip count while the y-axis depicts BARRIERFINDER's middle-end runtime in seconds averaged over 10 consecutive runs. Since performance variability is small (see standard deviations in Tab. 1), other runtime statistics are omitted to save space. Note that results for IR are provided only if BARRIERFINDER does not exhaust the 16GB memory under that configuration.

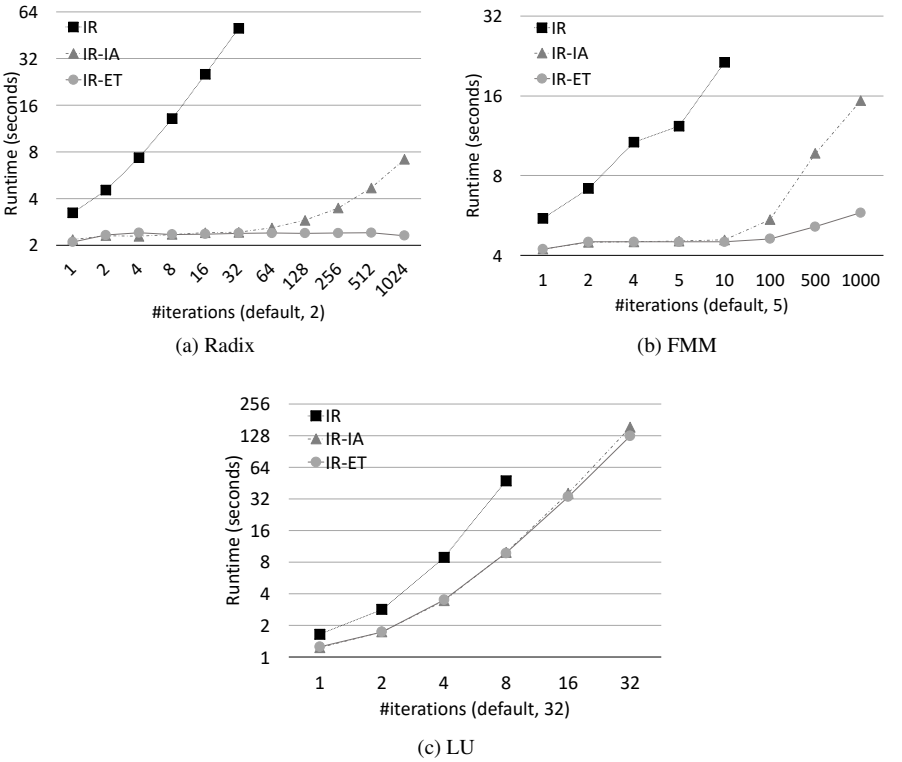
The results show IA and ET are the enabling techniques for exploring sync regions in loops. As we see in these figures, ET and IA significantly improve BARRIERFINDER's efficiency compared to IR only. Such a performance improvement is critical for LU, which otherwise cannot be explored without reduction on its loop trip count when only IR is enabled. Also, IA and ET enable BARRIERFINDER to explore two orders of magnitude more iterations than IR-only for Radix and FMM.

4.4 Effectiveness Results on Synthetic Benchmarks

Our synthetic benchmark suite contains eight variants of barriers and one allAB. As mentioned in Section 4.2, most of them are flawed implementations with different root causes, demonstrating the challenges that developers may face in practice. The expected trace patterns without sync region IDs are WRRWRR and WWR for barriers and allAB, respectively. BARRIERFINDER exits once a violation to either pattern or a deadlock is found.

Tab. 2 shows the effectiveness results of BARRIERFINDER on our synthetic benchmark suite. "Deadlock" indicates if a deadlock occurs; "Violation trace" is the first trace violating expected trace patterns while "Violation sequence ID" shows how many valid traces have been generated before the violation trace; "Atomic" indicates whether or not the barrier counter, e.g., `gcount` in Fig. 2 is accessed and checked atomically, and "Description" provides a short description for each benchmark's detection result.

We make the following observations from results in Tab. 2:



(a) Radix

(b) FMM

(c) LU

Fig. 7: The effect of interleaving avoidance (IA) and early loop termination (ET) under different iteration counts

① BARRIERFINDER has neither false positives nor false negatives for the synthetic application suite ("Actual" and "Characterized" are the same), since it enumerates all non-equivalent feasible interleavings.

② BARRIERFINDER reliably differentiates correct barrier implementations from wrong ones. Barriers 1 and 2 are correct while others have different problems. For a violation, BARRIERFINDER not only reports the violation trace but also produces the contexts, e.g., thread scheduling status and call stacks, which helps programmers understand the root causes. As shown by "Description", incorrect barriers encounter different problems.

③ BARRIERFINDER recognizes allAB, which can be regarded as a different variant of barriers. This also shows the potential for further generalization of the framework to other ad hoc synchronizations.

Table 2: Overall results of BARRIERFINDER on the synthetic application suite. Numbers in traces are sync region IDs.

ID	Actual	Characterized	Deadlock	Violation trace	Violation sequence ID	Atomic	Description
1	barrier	barrier	No	N/A	N/A	Yes	A reusable barrier
2	barrier	barrier	No	N/A	N/A	Yes	A reusable barrier
3	bad barrier	bad barrier	Yes	11WRR	3320	Yes	Both threads are blocked at the second invocation
4	bad barrier	bad barrier	No	11WR2RR2WR	0	Yes	Pattern violation
5	bad barrier	bad barrier	No	11WR2RR2WR	0	Yes	Pattern violation
6	bad barrier	bad barrier	Yes	11WR2R2WRW	18	No	One thread goes through the second invocation, the other is blocked at the second invocation
7	bad barrier	bad barrier	Yes	11WR2	0	No	One thread is blocked at the first invocation, the other is blocked at the second invocation
8	bad barrier	bad barrier	Yes	11WR2	0	Yes	same as above
9	allAB	allAB	No	N/A	N/A	N/A	allAB

4.5 Generalization to Any Number of Threads

BARRIERFINDER successfully enumerates the interleaving space of ad hoc barriers when the number of participating threads is small, which is critical for automatic recognition of synchronizations and understanding of their correctness. To go beyond that, we give an inductive proof over the number of participating threads, n. Others can use our proof as a template to generalize the results from BARRIERFINDER that an ad hoc synchronization is a barrier with two participating threads to any number of participating threads, even for other synchronization constructs. We highlight that the base case ($n = 2$) for our inductive proof is proved by BARRIERFINDER. Without BARRIERFINDER, such a manual inductive proof would be extremely tedious, if not infeasible.

Consider the following invariants for the algorithm in Fig. 2: (1) At line 28, which we refer to in the following as the program blocking point, P_b, the invariant $I_b(i)$: $newcount = i$ holds for all threads $1..n-1$ and these threads will busy wait at P_b as long as $gsense = lsense$. (2) At line 23, $I_r(i)$: $newcount = i$ holds for thread $i = n$, and the postcondition of I_r is $gsense \neq lsense$, which will subsequently cause threads $1..n-1$ to proceed past P_b by exiting the loop. In conjunction, I_b and I_r establish the barrier semantics, where $1..n-1$ threads wait until thread n releases the others and then proceeds to P_b itself, where it does not enter the loop as $gsense \neq lsense$. Notice that *lcount* and *newcount* are local variables with thread-specific values while *gsense* is a global variable shared between threads.

Base: Let us assume that $lsense = 1$. For $n = 2$, thread 1 is first to successfully execute CmpXchg (such that $updatecount = oldcount$), i.e., its $newcount = 1 = I_b(1)$ (as *gcount* is incremented by 1) such that it proceeds to P_b eventually. Thread 2 will succeed in CmpXchg later so that its $newcount = 2 = n = I_r(2)$, i.e., it will get to P_r and set *gsense* to the inverse of *lsense*. This releases thread 1, which will eventually

proceed past P_b and allows thread 2 to bypass the loop at P_b so that both thread exit the barrier. (The argument for $lsense = -1$ is symmetrical with decrements over *gcount*, where thread 2 eventually reaches P_r with $newcount = 0$.) In fact, BARRIERFINDER has already proved that these invariants hold as part of the states at P_b, P_r considered during execution interleaving, including the correct barrier semantics of leaving only after all threads have arrived, by exhaustive state enumeration.

Hypothesis: For n threads, let us assume that $I_b(i)$ holds for all $i = 1..n-1$ threads and I_r holds for thread n, including correct barrier semantics upon proceeding past these program points.

Step: For $n+1$ threads (and $gsense = 1$), consider two cases.

(1) Let the last one to succeed with the CmpXchg be thread $n+1$. For threads $1..n-1$, the hypothesis established $I_b(n)$ at P_b with increasing *newcount* values as $gsense = 1$. Thread n is now the second to last one to succeed in CmpXchg, so $newcount = n = I_r(n)$, which causes this thread to bypass P_r and proceed toward P_b, where it would busy wait due to $gsense = lsense$. And for thread $n+1$, as the last one to succeed in CmpXchg, $newcount = n = I_b(n+1)$ with $lsense = gsense = 1$, reaching P_r to invert *gsense* before reaching P_b without entering the loop $gsense \neq lsense$.

(2) Let thread $n+1$ be any base the last thread to succeed in CmpXchg. Without loss of generality, let thread $n+1$ succeed as the m-th thread in CmpXchg. Then there are threads $i = 1..m-1$ who succeeded in CmpXchg before and, with increasing *lsense*, are proceeding to P_b under $I_b(i)$ by the hypothesis. For thread $n+1$, $newcount = m = I_b(m)$, so it proceeds toward P_b. Threads $j = m+1..n$ succeed in CmpXchg in the respective indexed order next, i.e., their respective $newcount = j = I_r(j)$. The last one to succeed in CmpXchg, say thread l, has $newcount = n+1 = I_r(n+1)$ and proceeds to P_r inverting *gsense* and then to P_b bypassing the loop as per hypothesis.

This establishes the correct barrier semantics upon continuing past P_b for all threads. The cases for $gsense = -1$ are symmetrical (with decrements per thread succeeding in CmpXchg). Furthermore, alternating *gsense* signs upon successive barriers of $n+1$ threads establish the same barrier semantics as for n threads, i.e., only after P_r is reached by the last thread in the previous barrier may all threads proceed to enter the next barrier, where they then enter in increasing/decreasing *newcount* order for $gsense = 1/gsense = -1$. Any thread still at P_b of the previous barrier may proceed as their local $lsense \neq gsense$ while other threads already in the next barrier set *lsense* to *gsense* (line 13), so that they eventually spin in line 28 at P_b, other than the last thread.

4.6 Limitations and Future Work

We have open-sourced a software package Wang (2020) for the reproduction of BARRIERFINDER and for improvement by the research community. Currently, we make several assumptions that lead to the following limitations in BARRIERFINDER, which we leave as future work:

1. BARRIERFINDER currently only supports the automatic detection and correctness verification of counter-based ad hoc barriers and detection of the allAB re-

lationships. We plan to enable support of other types of barriers, like tree-based ones, by extending BARRIERFINDER with more pre-defined patterns and adding necessary techniques.
2. BARRIERFINDER works with global thread-pool based programs. The master thread spawns multiple child threads, which participate in the same barriers. BARRIERFINDER cannot handle multi-threaded programs with multiple thread pools or the case that only part of all threads participate in the same barriers. This may require manual reduction on the scale or the concurrency model of the source programs before being applied to BARRIERFINDER.
3. BARRIERFINDER's overhead reduction optimizations we proposed so far may only work with other barrier implementations and ad hoc synchronizations after modification. Applicability and scalability of these techniques on new applications, e.g, other types of ad hoc synchronizations mixed with ad hoc barriers, still need further assessment.

5 Related work

5.1 Synchronization Characterization and Detection

Several empirical studies related to synchronizations have been performed. Xiong et al. (Xiong et al. 2010) characterize ad hoc synchronizations of representative open-source applications and find they are pervasive. Pinto et al. (Pinto et al. 2015; Wu et al. 2016) survey real-world C++ and Java programs to assess how programs are synchronized in practice with concurrent language features, concurrent libraries, or concurrent data structures. Concurrent bug studies (Farchi et al. 2003; Lu et al. 2008; Zhang et al. 2011) try to characterize the pattern of concurrency bugs to facilitate their detection. Gu et al. (Gu et al. 2015) investigate how programmers change program synchronizations and their relation to concurrency bugs. The results of these studies motivate us to work on accurate synchronization understanding and guide the design of our approach.

Specifically, on ad hoc synchronizations, existing techniques (Jannesari and Tichy 2010, 2014; Tian et al. 2008, 2009; Xiong et al. 2010; Yin 2013; Yuan et al. 2013) use either static or dynamic approaches to detect sync pairs. We proceed further to detect complete synchronizations and recognize enforced synchronization relationships.

5.2 Barrier Analysis

Kamil et al. (Kamil and Yelick 2006) propose interprocedural concurrency analysis for single program multiple data programs written in a Java dialect, Titanium, which statically guarantees that all threads reach the same sequence of textual barriers. Zhang et al. (Zhang et al. 2008) transform OpenMP programs into control flow graph with all OpenMP implied barriers, and then treat the barrier matching as a regular expression while parsing the CFG. Their approach can handle textually unaligned barriers. Our work does not assume any prior knowledge about barriers and tried to

recognize barriers with ad hoc implementations. As a result, BARRIERFINDER results can be used as a foundation for the aforementioned work of barrier analysis.

5.3 Runtime Invariant Detection

Our approach recognizes ad hoc barriers by mining execution traces for temporal invariants. Invariant mining is a technique pioneered by Daikon (Ernst et al. 2007), 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. (Beschastnikh et al. 2011) propose techniques to mine temporal invariants based on partially ordered logs, and CSight (Beschastnikh et al. 2014) further uses temporal invariants to model concurrent systems. CloudSeer (Yu et al. 2016) 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.

5.4 Interleaving Reduction

Due to the exponential explosion of thread interleavings in multithreaded programs, there has been a large body of work on reducing the interleaving space and optimizing interleaving exploration in testing and verification. Limiting scheduling points, threads, or sync variables is a common idea to reduce the interleaving space. Musuvathi and Qadeer (Musuvathi and Qadeer 2007) propose an iterative context-bounding technique that limits the number of preempting context switches. Blum and Gibson (Blum and Gibson 2016) propose on-the-fly adjustment of preemption points. Bindal et al. (Bindal et al. 2013) propose a variable and thread bounding technique. In our work, the slicing-based scope reduction leverages the similar idea by identifying and using high-level sync regions as the scheduling scope. In addition to constraining the space, some work focuses on finding equivalent interleavings to avoid exhaustive interleaving enumeration. Mazurkiewicz equivalence (Mazurkiewicz 1987) is a widely accepted equivalence class, and there are also variations (Chalupa et al. 2017; Wang et al. 2009). In the future, we can incorporate these interleaving reduction techniques while extending our framework for other more complex ad hoc synchronizations.

5.5 Program Slicing

Classic program slicing (Weiser 1981, 1984) is based on program's control flow graph of sequential programs and is known to produce unnecessarily large slices due to the calling context problem (Horwitz et al. 1990), which has been addressed (Gallagher 2004) and is functionally equivalent to more recent slicing techniques (Horwitz et al. 1990) based on program dependency graphs (PDG). In practice, the wide acceptance of pointers in programming languages, such as C/C++, slices inflate even

more because of a conservative interpretation of imprecise pointer analyses (Andersen 1994; Shapiro and Horwitz 1997; Steensgaard 1996). Anderson's algorithm (Andersen 1994) is known to be more precise than others (Shapiro and Horwitz 1997; Steensgaard 1996) and popular in practice. Moreover, as Hind at al. (Hind and Pioli 2000) show, adding context-sensitivity and flow-sensitivity results in little improvement in precision. In our work, BARRIERFINDER's front end is based on LLVM-Slicer (Slaby 2015), implementing Anderson's algorithm with field-sensitivity. Thus, it can perform slicing on fields of a structure. We notice there are many occasions that a sliced program contains time-consuming computation, which can be omitted in the slice. One may try to devise more precise pointer analysis techniques (Hardekopf and Lin 2007), fixing the calling context problem or employing PDG-based slicers.

The slicing of concurrent programs is more challenging (Giffhorn and Hammer 2009; Krinke 2003; Nanda and Ramesh 2006). Precision depends on the model of concurrency, i.e., the program synchronization structure. Our goal is to facilitate the understanding of the model of concurrency by inferring the semantics of synchronization constructs used. Instead of relying on these concurrent program slicing approaches, we enhance the original slicing algorithm in LLVMSlicer (Slaby 2015) by constraining the underlying concurrency model to the work-crew model (corporation 1998) based on the Pthread standard. In the future, we may resort to more general algorithms (Krinke 2003; Nanda and Ramesh 2006).

5.6 Related Program Development Tools

Various program development tools, such as data race detectors (Bessey et al. 2010; Lee et al. 2012; Raman et al. 2012; Sadowski and Yi 2014; Surendran et al. 2014), atomicity violation detectors (Lucia et al. 2010; Park et al. 2009), order violation detectors (Zhang et al. 2010), synchronization-oriented performance profilers (Chen and Stenstrom 2012; Yu and Pradel 2016) and concurrent program slicers (Krinke 2003; Nanda and Ramesh 2006), depend on the understanding of the input program synchronization structure for their accuracy or performance. Such critical information is usually obtained by may-happen-in-parallel analysis (Chen et al. 2012; Di et al. 2015; Li and Verbrugge 2005), which further depends on the understanding of both standard and custom synchronizations. We believe SynCat's capability of automatically understanding custom synchronizations is complementary and fundamental to all the aforementioned dependent analysis and development tools.

6 Conclusion

This paper contributes BARRIERFINDER, a pipelined framework to automate the recognition of complex ad hoc synchronizations that realize barriers. During compile time, BARRIERFINDER applies program slicing to reduce irrelevant computation and develops a novel approach to detect synchronization boundaries for reducing the scope of interleaving enumerations. During runtime, a sequence of interleaving space reduction techniques greatly shrinks the exponential interleaving space into a linear

one, in terms of number of barriers in a program. Various intrusive interpretation-based optimizations further improve the execution efficiency. BARRIERFINDER addresses the space-explosion problem with these techniques when there are two participating threads, which establishes a base case for an inductive proof to generalize the result for any number of threads. The experimental evaluation shows that BARRIERFINDER is able to detect barriers in six SPLASH2 benchmarks efficiently. To our knowledge, BARRIERFINDER is the first tool that can detect ad hoc barriers as a whole synchronization constructs. We further assess BARRIERFINDER with a set of synthetic benchmarks most of which are incorrect counter-based barrier implementations. Our evaluation demonstrates that BARRIERFINDER is capable of detecting different implementation errors, verify the correctness of counter-based barriers, and characterizing other ad hoc synchronizations like allAB. We believe BARRIERFINDER contributes to the fundamental problem of accurate program synchronization understanding and has the potential to improve a spectrum of program analysis and development tools.

References

Andersen L. O. (1994) Program analysis and specialization for the c programming language. PhD thesis, University of Cophenhagen

Beschastnikh I., Brun Y., Ernst M. D., Krishnamurthy A., Anderson T. E. (2011) 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, ACM, New York, NY, USA, SLAML '11, pp. 3:1–3:10, DOI 10.1145/2038633.2038636, URL http://doi.acm.org/10.1145/2038633.2038636

Beschastnikh I., Brun Y., Ernst M. D., Krishnamurthy A. (2014) Inferring models of concurrent systems from logs of their behavior with csight. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp. 468–479, DOI 10.1145/2568225.2568246, URL http://doi.acm.org/10.1145/2568225.2568246

Bessey A., Block K., Chelf B., Chou A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D. (2010) A few billion lines of code later: Using static analysis to find bugs in the real world. Commun ACM 53(2):66–75, DOI 10.1145/1646353.1646374, URL http://doi.acm.org/10.1145/1646353.1646374

Bindal S., Bansal S., Lal A. (2013) Variable and thread bounding for systematic testing of multithreaded programs. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2013, pp. 145–155, DOI 10.1145/2483760.2483764, URL http://doi.acm.org/10.1145/2483760.2483764

Blum B., Gibson G. (2016) Stateless model checking with data-race preemption points. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA, OOPSLA 2016, pp. 477–493, DOI 10.1145/2983990.2984036, URL http://doi.acm.org/10.1145/2983990.2984036

Bucur S., Ureche V., Zamfir C., Candea G. (2011) Parallel symbolic execution for automated real-world software testing. In: Proceedings of the sixth conference on Computer systems, ACM, pp. 183–198

Chalupa M., Chatterjee K., Pavlogiannis A., Sinha N., Vaidya K. (2017) Data-centric dynamic partial order reduction. Proc ACM Program Lang 2(POPL):31:1–31:30, DOI 10.1145/3158119, URL http://doi.acm.org/10.1145/3158119

Chen C., Huo W., Feng X. (2012) Making it practical and effective: Fast and precise may-happen-in-parallel analysis. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ACM, New York, NY, USA, PACT '12, pp. 469–470, DOI 10.1145/2370816.2370900, URL http://doi.acm.org/10.1145/2370816.2370900

Chen G., Stenstrom P. (2012) Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, SC '12, pp. 71:1–71:11, URL http://dl.acm.org/citation.cfm?id=2388996.2389093

corporation I. (1998) Software models for multithreaded programming. http://www-01.ibm.com/software/network/dce/library/publications/dceaix_22/a3u2j/A3U2JM53.HTM

Cui H., Simsa J., Lin Y.-H., Li H., Blum B., Xu X., Yang J., Gibson G. A., Bryant R. E. (2013) Parrot: A practical runtime for deterministic, stable, and reliable threads. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA, SOSP '13, p. 388–405, DOI 10.1145/2517349.2522735, URL https://doi.org/10.1145/2517349.2522735

Das M., Southern G., Renau J. (2015) Section-based program analysis to reduce overhead of detecting unsynchronized thread communication. ACM Trans Archit Code Optim 12(2):23:23:1–23:23:26, DOI 10.1145/2766451, URL http://doi.acm.org/10.1145/2766451

Di P., Sui Y., Ye D., Xue J. (2015) Region-based may-happen-in-parallel analysis for c programs. In: Parallel Processing (ICPP), 2015 44th International Conference on, pp. 889–898, DOI 10.1109/ICPP.2015.98

Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S., Xiao C. (2007) The daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1):35–45

Farchi E., Nir Y., Ur S. (2003) Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, IEEE, pp. 7–pp

Gallagher K. B. (2004) Some notes on interprocedural program slicing. In: Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on, pp. 36–42, DOI 10.1109/SCAM.2004.21

Giffhorn D., Hammer C. (2009) Precise slicing of concurrent programs. Automated Software Engg 16(2):197–234, DOI 10.1007/s10515-009-0048-x, URL http://dx.doi.org/10.1007/s10515-009-0048-x

Gu R., Jin G., Song L., Zhu L., Lu S. (2015) What change history tells us about thread synchronization. In: Proceedings of the 2015 10th Joint Meeting on Foundations of

Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp. 426–438, DOI 10.1145/2786805.2786815, URL http://doi.acm.org.prox.lib.ncsu.edu/10.1145/2786805.2786815

Hardekopf B., Lin C. (2007) The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '07, pp. 290–299, DOI 10.1145/1250734.1250767, URL http://doi.acm.org/10.1145/1250734.1250767

Herb S. (2005) The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm

Hind M., Pioli A. (2000) Which pointer analysis should i use? In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA '00, pp. 113–123, DOI 10.1145/347324.348916, URL http://doi.acm.org/10.1145/347324.348916

Horwitz S., Reps T., Binkley D. (1990) Interprocedural slicing using dependence graphs. ACM Trans Program Lang Syst 12(1):26–60, DOI 10.1145/77606.77608, URL http://doi.acm.org/10.1145/77606.77608

Jannesari A., Tichy W. F. (2010) Identifying ad-hoc synchronization for enhanced race detection. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, pp. 1–10

Jannesari A., Tichy W. F. (2014) Library-independent data race detection. IEEE Transactions on Parallel and Distributed Systems 25(10):2606–2616

Jin G., Song L., Zhang W., Lu S., Liblit B. (2011) Automated atomicity-violation fixing. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '11, pp. 389–400, DOI 10.1145/1993498.1993544, URL http://doi.acm.org/10.1145/1993498.1993544

Jin G., Zhang W., Deng D., Liblit B., Lu S. (2012) Automated concurrency-bug fixing. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'12, pp. 221–236, URL http://dl.acm.org/citation.cfm?id=2387880.2387902

Kamil A., Yelick K. (2006) Concurrency analysis for parallel programs with textually aligned barriers. In: Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing, Springer-Verlag, Berlin, Heidelberg, LCPC'05, pp. 185–199, DOI 10.1007/978-3-540-69330-7_13, URL http://dx.doi.org/10.1007/978-3-540-69330-7_13

Krinke J. (2003) Context-sensitive slicing of concurrent programs. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE-11, pp. 178–187, DOI 10.1145/940071.940096, URL http://doi.acm.org/10.1145/940071.940096

Lee D., Chen P. M., Flinn J., Narayanasamy S. (2012) Chimera: Hybrid Program Analysis for Determinism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '12, pp. 463–474, DOI 10.1145/2254064.2254119, URL http:

//doi.acm.org/10.1145/2254064.2254119

Li L., Verbrugge C. (2005) A Practical MHP Information Analysis for Concurrent Java Programs, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 194–208. DOI 10.1007/11532378_15, URL http://dx.doi.org/10.1007/11532378_15

Lu S., Park S., Seo E., Zhou Y. (2008) Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XIII, pp. 329–339, DOI 10.1145/1346281.1346323, URL http://doi.acm.org/10.1145/1346281.1346323

Lucia B., Ceze L., Strauss K. (2010) Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, ISCA '10, pp. 222–233, DOI 10.1145/1815961.1815988, URL http://doi.acm.org/10.1145/1815961.1815988

Malkis A., Banerjee A. (2014) On automation in the verification of software barriers: Experience report. Journal of Automated Reasoning 52(3):275–329, DOI 10.1007/s10817-013-9290-9, URL https://doi.org/10.1007/s10817-013-9290-9

Mazurkiewicz A. (1987) Trace theory. In: Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency, Springer-Verlag New York, Inc., New York, NY, USA, pp. 279–324, URL http://dl.acm.org/citation.cfm?id=25542.25553

Musuvathi M., Qadeer S. (2007) Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '07, pp. 446–455, DOI 10.1145/1250734.1250785, URL http://doi.acm.org/10.1145/1250734.1250785

Nanda M. G., Ramesh S. (2006) Interprocedural slicing of multithreaded programs with applications to java. ACM Trans Program Lang Syst 28(6):1088–1144, DOI 10.1145/1186632.1186636, URL http://doi.acm.org/10.1145/1186632.1186636

Park S., Lu S., Zhou Y. (2009) Ctrigger: Exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XIV, pp. 25–36, DOI 10.1145/1508244.1508249, URL http://doi.acm.org/10.1145/1508244.1508249

Pinto G., Torres W., Fernandes B., Castor F., Barros R. S. (2015) A large-scale study on the usage of java's concurrent programming constructs. J Syst Softw 106(C):59–81, DOI 10.1016/j.jss.2015.04.064, URL http://dx.doi.org/10.1016/j.jss.2015.04.064

Raman R., Zhao J., Sarkar V., Vechev M., Yahav E. (2012) Scalable and precise dynamic datarace detection for structured parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '12, pp. 531–542, DOI 10.1145/2254064.2254127, URL http://doi.acm.org/10.1145/2254064.2254127

Sadowski C., Yi J. (2014) How developers use data race detection tools. In: Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, ACM, New York, NY, USA, PLATEAU '14, pp. 43–51, DOI 10.1145/2688204.2688205, URL http://doi.acm.org/10.1145/2688204.2688205

Shapiro M., Horwitz S. (1997) Fast and accurate flow-insensitive points-to analysis. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '97, pp. 1–14, DOI 10.1145/263699.263703, URL http://doi.acm.org/10.1145/263699.263703

Slaby J. (2015) Llvm slicer. https://github.com/jirislaby/LLVMSlicer

Steensgaard B. (1996) Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '96, pp. 32–41, DOI 10.1145/237721.237727, URL http://doi.acm.org/10.1145/237721.237727

Surendran R., Raman R., Chaudhuri S., Mellor-Crummey J., Sarkar V. (2014) Test-driven repair of data races in structured parallel programs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '14, pp. 15–25, DOI 10.1145/2594291.2594335, URL http://doi.acm.org/10.1145/2594291.2594335

Sutter H., Larus J. (2005) Software and the concurrency revolution. Queue 3(7):54–62, DOI 10.1145/1095408.1095421, URL http://doi.acm.org/10.1145/1095408.1095421

Taubenfeld G. (2006) Synchronization Algorithms and Concurrent Programming. Prentice-Hall, Inc., Upper Saddle River, NJ, USA

Tian C., Nagarajan V., Gupta R., Tallam S. (2008) Dynamic recognition of synchronization operations for improved data race detection. In: Proceedings of the 2008 international symposium on Software testing and analysis, ACM, pp. 143–154

Tian C., Nagarajan V., Gupta R., Tallam S. (2009) Automated dynamic detection of busy–wait synchronizations. Software: Practice and Experience 39(11):947–972

Wang C., Chaudhuri S., Gupta A., Yang Y. (2009) Symbolic pruning of concurrent program executions. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '09, pp. 23–32, DOI 10.1145/1595696.1595702, URL http://doi.acm.org/10.1145/1595696.1595702

Wang T. (2020) Software package for barrierfinder reproduction. DOI 10.5281/zenodo.3902595, URL https://doi.org/10.5281/zenodo.3906920

Wang T., Yu X., Qiu Z., Jin G., Mueller F. (2019) Barrierfinder: Recognizing ad hoc barriers. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 323–327, DOI 10.1109/ICSME.2019.00049

Weiser M. (1981) Program slicing. In: Proceedings of the 5th international conference on Software engineering, IEEE Press, pp. 439–449

Weiser M. (1984) Program slicing. IEEE Transactions on Software Engineering SE-10(4):352–357, DOI 10.1109/TSE.1984.5010248

Woo S. C., Ohara M., Torrie E., Singh J. P., Gupta A. (1995) The splash-2 programs: Characterization and methodological considerations. In: Proceedings of

the 22Nd Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, ISCA '95, pp. 24–36, DOI 10.1145/223982.223990, URL http://doi.acm.org/10.1145/223982.223990

Wu D., Chen L., Zhou Y., Xu B. (2016) An extensive empirical study on c++ concurrency constructs. Information and Software Technology 76:1–18

Xiong W., Park S., Zhang J., Zhou Y., Ma Z. (2010) Ad hoc synchronization considered harmful. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'10, pp. 163–176, URL http://dl.acm.org/citation.cfm?id=1924943.1924955

Yin L. (2013) Effectively recognize ad hoc synchronizations with static analysis. In: International Workshop on Languages and Compilers for Parallel Computing, Springer, pp. 187–201

Yu T., Pradel M. (2016) Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp. 389–400, DOI 10.1145/2931037.2931070, URL http://doi.acm.org/10.1145/2931037.2931070

Yu X., Joshi P., Xu J., Jin G., Zhang H., Jiang G. (2016) Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS '16, pp. 489–502, DOI 10.1145/2872362.2872407, URL http://doi.acm.org/10.1145/2872362.2872407

Yuan X., Wang Z., Wu C., Yew P.-C., Wang W., Li J., Xu D. (2013) Synchronization identification through on-the-fly test. In: Proceedings of the 19th International Conference on Parallel Processing, Springer-Verlag, Berlin, Heidelberg, Euro-Par'13, pp. 4–15, DOI 10.1007/978-3-642-40047-6_3, URL http://dx.doi.org/10.1007/978-3-642-40047-6_3

Zhang W., Sun C., Lu S. (2010) 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, ACM, New York, NY, USA, ASPLOS XV, pp. 179–192, DOI 10.1145/1736020.1736041, URL http://doi.acm.org/10.1145/1736020.1736041

Zhang W., Lim J., Olichandran R., Scherpelz J., Jin G., Lu S., Reps T. (2011) Conseq: Detecting concurrency bugs through sequential errors. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XVI, pp. 251–264, DOI 10.1145/1950365.1950395, URL http://doi.acm.org/10.1145/1950365.1950395

Zhang Y., Duesterwald E., Gao G. R. (2008) Concurrency analysis for shared memory programs with textually unaligned barriers. In: Adve V., Garzarán M. J., Petersen P. (eds) Languages and Compilers for Parallel Computing, Springer-Verlag, Berlin, Heidelberg, pp. 95–109, DOI 10.1007/978-3-540-85261-2_7, URL http://dx.doi.org/10.1007/978-3-540-85261-2_7

Zhao Q., Qiu Z., Jin G. (2019) Semantics-aware scheduling policies for synchronization determinism. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, Association for Computing Machinery, New York, NY, USA, PPoPP '19, p. 242–256, DOI 10.1145/3293883.3295731, URL https://doi.org/10.1145/3293883.3295731