Concurrency Debugging with Differential Schedule Projections

NUNO MACHADO, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
DANIEL QUINTA, INESC-ID
BRANDON LUCIA, Carnegie Mellon University
LUÍS RODRIGUES, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

We present Symbiosis: a concurrency debugging technique based on novel *differential schedule projections* (DSPs). A DSP shows the small set of memory operations and dataflows responsible for a failure, as well as a reordering of those elements that avoids the failure. To build a DSP, Symbiosis first generates a *full, failing, multithreaded schedule* via thread path profiling and symbolic constraint solving. Symbiosis selectively reorders events in the failing schedule to produce a *nonfailing, alternate schedule*. A DSP reports the ordering and dataflow *differences* between the failing and nonfailing schedules. Our evaluation on buggy real-world software and benchmarks shows that, in practical time, Symbiosis generates DSPs that both isolate the small fraction of event orders and dataflows responsible for the failure and report which event reorderings prevent failing. In our experiments, DSPs contain 90% fewer events and 96% fewer dataflows than the full failure-inducing schedules. We also conducted a user study that shows that, by allowing developers to focus on only a few events, DSPs reduce the amount of time required to understand the bug's root cause and find a valid fix.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Concurrency, bug localization, constraint solving, differential schedule projection

ACM Reference Format:
Nuno Machado, Daniel Quinta, Brandon Lucia, and Luís Rodrigues. 2016. Concurrency debugging with differential schedule projections. ACM Trans. Softw. Eng. Methodol. 25, 2, Article 14 (April 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2885495

1. INTRODUCTION

Concurrent programming is the new norm, because it allows exploring the parallelism offered by recent multicore architectures. Unfortunately, concurrent and parallel programming are much more difficult than sequential programming. To exploit concurrency in multithreaded code, programmers need to reason about multiple threads of execution that interact by reading from and writing to shared memory locations. However, without proper synchronization, operations in different threads may nondeterministically adhere to different execution *schedules* and, consequently, produce different results. Although most schedules are correct, some *failing schedules* can lead

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project UID/CEC/50021/2013, and by a 2015 Google Faculty Research Award.
Authors' addresses: N. Machado and D. Quinta, INESC-ID, Office 612, Rua Alves Redol 9, 1000-029 Lisboa, Portugal; emails: nuno.machado@tecnico.ulisboa.pt, danielribeiro.mail@gmail.com; B. Lucia, CIC 4th Floor, 101A, Robert Mehrabian Collaborative Innovation Center (CIC), Carnegie Mellon University, 4720 Forbes Avenue, Pittsburgh, PA 15213; email: blucia@cmu.edu; L. Rodrigues, INESC-ID, Office 508, Rua Alves Redol 9, 1000-029 Lisboa, Portugal; email: ler@tecnico.ulisboa.pt.

DOI: http://dx.doi.org/10.1145/2885495

to undesirable behavior, like a crash or data corruption. A failing schedule is the result of a *concurrency bug*, which a mistake in the code that incorrectly permits an ordering of thread operations that leads to a failure.

Eliminating concurrency bugs is extremely difficult. Failing schedules may manifest rarely and reproducing them is often difficult. Prior work has addressed reproducibility with a number of different strategies, including *record and replay* (R&R) (both order based (Huang et al. [2010], Yang et al. [2011], and Jiang et al. [2014]) and search based (Machado et al. [2012], Zhou et al. [2012], and Huang et al. [2013])) and *deterministic execution* (Olszewski et al. [2009], Berger et al. [2009], and Devietti et al. [2009]). These techniques allow the developer to observe a failing execution multiple times, but simply reproducing a failure may provide no insight into its cause. The key to debugging a concurrency bug is understanding the failure's *root cause*, that is, the set of event orderings that are *necessary* for failure. Although the number of events that comprise a root cause is typically small [Burckhardt et al. 2010], it is often unclear which events in a full schedule to focus on. Any operation in any thread may have led to the failure and blindly analyzing a full schedule is a metaphorical search for a *needle in a haystack*. On the other hand, even if the programmer finds the root cause, they still face the difficult task of understanding and changing the code to prevent the problematic thread interleaving from happening in the future.

We present Symbiosis, a system that helps finding and understanding a failure's root cause, as well as fixing the underlying bug. Figure 1 presents a schematic view of our system. Symbiosis first collects single-threaded path profiles from a concrete, failing execution. The profiles guide a symbolic execution, yielding per-thread symbolic event traces compatible with the failure. These are then used to generate a Satisfiability Modulo Theory (SMT) formulation, the solution to which represents a multithreaded failing schedule. To prune irrelevant events from the failing schedule, Symbiosis generates an *unsatisfiable* SMT formulation encoding the failing schedule, but the absence of the failure. As a result, the SMT solver reports a subset of constraints that conflict in the unsatisfiable SMT formulation; their corresponding event orderings are necessary for the failure, and form the pruned *root-cause schedule*. The root-cause schedule is used in another SMT formulation to compute *nonfailing, alternative schedules* that comprise reorderings of the root-cause schedule's events. Symbiosis enhances the debugging utility of the root-cause schedule by reporting *only the important ordering and dataflow differences* between failing and nonfailing schedules. These differences are dataflows between operations in the failing execution that do not occur in the correct execution and vice versa. We call the output of our novel debugging approach a *differential schedule projection* (DSP).

DSPs simplify debugging for two main reasons. First, by showing only what differs between a failing and nonfailing schedule, the programmer is exposed to a very small number of relevant operations, rather than a full schedule. Second, DSPs illustrate not only the failing schedule but also the way execution *should* behave, if not to fail. Seeing the different event orders side by side helps understand the failure and, often, how to fix the bug. Although Symbiosis is only able to isolate the failure for a particular execution path, we still believe DSPs to be a significant improvement with respect to more traditional debugging approaches, such as *cyclic debugging* (i.e., iteratively reexecute a program's failing execution in an attempt to understand the bug and narrow its root cause).

Critically, Symbiosis produces a DSP from a *single* failing schedule, enabling its use for failures observed rarely (i.e., in production). This contrasts to prior work in [Lucia and Ceze 2013] and [Kasikci et al. 2015] that relies on statistical inference and, therefore, needs to capture information from a significant amount of failing executions in order to isolate the bug's root cause effectively. Our evaluation in Section 5 shows that

DSPs have, on average, 90% fewer events than full schedules and shows qualitatively, with case studies, that DSPs help understand failures and fix bugs. Furthermore, we conducted a user study with 48 participants to further support the claim that DSPs allow for faster bug diagnosis.

To summarize, our contributions are as follows:

—An SMT constraint formulation, based on the computed failing schedule, that identifies the subschedule that is a failure's root cause.
—A heuristic, based on SMT constraint formulations, that systematically varies the order of root-cause events to find alternative nonfailing schedules similar to the original failing schedule.
—A novel *differential schedule projection* methodology that isolates important control and dataflow changes between failing and nonfailing schedules computed by Symbiosis.
—An implementation of Symbiosis for C/C++ and Java and an evaluation, showing the debugging efficacy and efficiency of Symbiosis.
—A user study that provides evidence that DSPs allow developers to diagnose concurrency failures faster than with full failing schedules.

The rest of the article is organized as follows. Section 2 overviews the background concepts most related to our work. Section 3 describes the Symbiosis system in detail, namely its components and how it operates to produce DSPs. Section 4 reports the implementation details. Section 5 presents the results of both the experimental evaluation and the user study and discusses the main findings. Finally, Section 6 reviews the related work.

2. BACKGROUND

Symbiosis helps with concurrency debugging by leveraging prior work on symbolic execution and SMT solving. This section briefly reviews these topics.

Concurrency Bugs. Concurrency bugs are errors in code that permit multithreaded schedules that lead to a failure. Concurrency bugs have been studied extensively in the literature [Lu et al. 2006; Flanagan et al. 2008; Zhang et al. 2010, 2011; Engler and Ashcraft 2003; Savage et al. 1997; Flanagan and Freund 2009; Lucia et al. 2011; Lucia and Ceze 2013; Lucia et al. 2008]. Data races [Savage et al. 1997; Flanagan and Freund 2009; Engler and Ashcraft 2003], atomicity violations [Lu et al. 2006; Flanagan et al. 2008; Lucia et al. 2008], ordering violations [Lu et al. 2008; Lucia and Ceze 2009; Park et al. 2010; Zhang et al. 2010, 2011] and deadlocks [Engler and Ashcraft 2003; Zamfir and Candea 2010b] are different types of concurrency bugs studied by prior work. These bugs vary in their mechanism and result. For example, while data races may lead to violations of sequential consistency [Lamport 1979], atomicity violations may lead to unserializable behavior of atomic regions. We defer to the literature for a detailed discussion of these bug types and their failure modes. Instead, we just emphasize that they share the following important characteristic: They lead to a failure when they permit operations in different threads to execute in an order that should be forbidden. Symbiosis attacks the debugging problem by identifying such incorrect operation orderings that constitute the root cause of a failure.

Symbolic Execution. *Symbolic execution* [King 1976] explores the space of possible executions of a program by emulating or directly executing its statements. During symbolic execution, some variables, such as inputs, have *symbolic* values. A symbolic value represents a set of possible concrete values. Assignments to and from symbolic variables and operations involving symbolic variables produce results that are also

symbolic. When an execution reaches a branch dependent on a symbolic variable, it spawns two identical copies of the execution state—one in which the branch is taken and one in which the branch is not taken. Spawned copies continue independently along these different *paths* and the process repeats for every new symbolic branch. Each path has a *path constraint*, encoding all branch outcomes on that path. Thus, the path constraint determines possible concrete values for symbolic variables that lead execution down a particular path. To simplify complex symbolic constraints, some systems employ *concolic execution* [Sen et al. 2005] which uses both concrete and symbolic values for variables.

As we describe in Section 3.3, Symbiosis uses symbolic execution to find a path in each thread that leads to a failure. Concretely, Symbiosis treats *shared variables* as symbolic, because they might be modified nondeterministically by any thread during a multithreaded execution. In our Symbiosis prototype for C/C++ programs, we manually identify shared variables, whereas, in our prototype for Java applications, this is done via static analysis. We discuss the impact of these choices in Section 4.

SMT Solvers. An SMT solver is a tool that, given a formula over variables, finds a satisfying assignment of the variables or reports that it is unsatisfiable. SMT is based on Boolean satisfiability (SAT). However, SMT is more expressive than SAT for, for example, handling arithmetic. SAT and SMT are NP-complete, but decades of research have produced solvers (e.g., Z3 De Moura and Bjørner [2008]) that practically solve large problems. Practical SMT has found use in many areas: hardware [Emmer et al. 2010] and software verification [Qadeer 2009], program analysis [Lahiri and Qadeer 2008], and test generation [Tillmann and De Halleux 2008].

When an SMT formula is unsatisfiable, some SMT solvers [De Moura and Bjørner 2008] are able to *explain why* by reporting which constraints conflict in an unsatisfiability core, or *UNSAT Core*. BugAssist [Jose and Majumdar 2011] pioneered the use of the UNSAT core to help isolate errors in sequential programs. In this work, Symbiosis makes novel use of this idea to debug concurrency errors and reduce the information it must analyze when building differential schedule projections.

Computing Schedules with Symbolic Execution and SMT Solvers. Symbiosis requires a schedule from a failing execution in order to isolate the root cause of a concurrency bug. Although the technique used to obtain the failing schedule is orthogonal to Symbiosis, in this work, we follow the approach proposed by CLAP [Huang et al. 2013]. CLAP is a system for reproducing concurrency failures, which links concurrency, SMT, and symbolic execution.

Symbiosis builds on the following CLAP features: independent tracking of thread control flow, use of guided symbolic execution, and resorting to SMT constraints to compute a failing schedule (see Section 3). This approach uses concrete, per-thread path profiles to guide a symbolic execution of the program and generate per-thread symbolic traces. Then it encodes a set of SMT formulae that constrain the variables contained in all threads' sequential symbolic traces, along with additional constraints encoding interthread dataflow and synchronization. When solved by an off-the-shelf SMT solver, the formulation yields a failing, multithreaded schedule.

Like CLAP, Symbiosis can *also* reproduce bugs by adding constraints corresponding to a failure's manifestation (Section 3.3). However, despite these similarities, Symbiosis differs fundamentally from CLAP in its purpose and mechanisms. CLAP aims at replaying failing schedules, while Symbiosis focus on isolating the root cause of concurrency bugs. Moreover, unlike CLAP, Symbiosis produces precise root-cause subschedules and alternate, nonfailing subschedules. Symbiosis's subschedule reports include many fewer operations than full schedules and do not require examining the whole schedule to find the few operations involved in the failure, as is the case with CLAP.

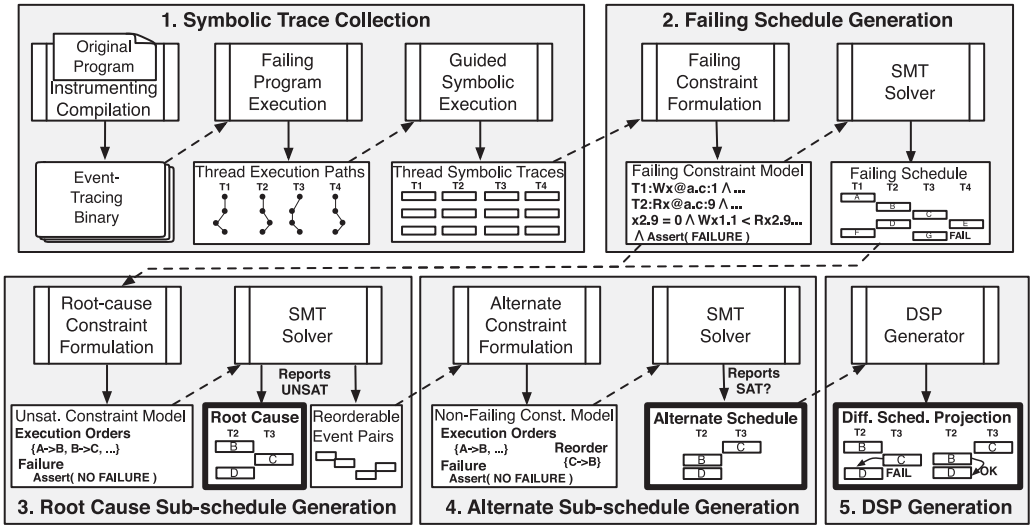

Fig. 1. Overview of Symbiosis. Boxes at the top represent processes. Boxes at the bottom represent inputs and outputs of processes. Dashed arrows denote an input relationship and solid arrows denote and output relationship. Bold boxes represent the final outputs of Symbiosis.

Symbiosis's subschedule reports are the foundation of differential schedule projections, a new debugging technique not explored by CLAP. Symbiosis's mechanism for computing subschedules based on the SMT solver's UNSAT core is novel, as is Symbiosis's technique for reordering event pairs to compute alternate, nonfailing subschedules.

3. SYMBIOSIS

In this section, we start by presenting an overview of Symbiosis and how it operates. Then we describe in detail each component of our system.

3.1. Overview

Symbiosis is a technique for concisely reporting the root cause of a failing multithreaded execution, alongside a nonfailing, alternate execution of the events that make up the root cause. Symbiosis produces *differential schedule projections*, which reveal bugs' root causes and aid in debugging. Symbiosis has five phases (see Figure 1) as follows:

1. Symbolic trace collection. In a concrete failing program run, Symbiosis traces the sequence of basic blocks executed by each thread independently. The per-thread path profiles are used to guide symbolic execution, producing a set of per-thread traces with symbolic information (e.g., path conditions and read-write accesses to shared variables).

2. Failing schedule generation. Symbiosis produces an SMT formula that corresponds to the symbolic execution trace. The formula includes constraints that represent each thread's path, as well as the failure's manifestation, memory access orderings, and synchronization orderings. The solution to the SMT formula corresponds to a complete, failing, multithreaded execution. In other words, this solution specifies the ordering of events that triggers the error.

3. Root cause subschedule generation. Symbiosis produces an SMT formula corresponding to the symbolic trace, but specifies that the execution should not fail, by negating the failure condition. Combined with the constraints representing the order of

events in the full, failing schedule, the SMT instance is unsatisfiable. The SMT solver produces an UNSAT core that contains the constraints representing the execution event orders that conflict with the absence of the failure. Those event orders are necessary for the failure to occur, that is, the failure's root-cause subschedule.

4. Alternate subschedule generation. Symbiosis examines each pair of events from different threads in the root cause. For each pair, Symbiosis produces a new SMT formula, identical to the one used to find the root cause, but with constraints implying the ordering of the events in the pair reversed. When Symbiosis finds an instance that is satisfiable, the corresponding schedule is very similar to the failing schedule[1] but does not fail. Symbiosis reports the alternate, nonfailing schedule that is identical to the failing schedule but with the pair of events reordered.

The experimental results in Section 5 indicate that this technique is effective and that Symbiosis was able to find a nonfailing schedule by reordering fewer than 10 pairs of events for 10 of 13 test cases.

5. Differential schedule projection generation. Symbiosis produces a differential schedule projection by comparing the failing schedule and the alternate, nonfailing subschedule. The DSP shows how the two schedules differ in the order of their events and in their dataflow behavior. Additionally, as the reordered pair from the alternate nonfailing schedule eliminates an event order necessary for the failure to occur, it can be leveraged by a dynamic failure avoidance system [Lucia and Ceze 2013] to prevent future failures.

To better illustrate the main concepts of Symbiosis, we use a running example that consists of the modified version of `pfscan` file scanner studied in prior work [Elmas et al. 2013]. A slightly simplified snippet of the program's code is depicted in Figure 2(a). The program uses three threads. The first thread enqueues elements into a shared queue. The two other threads attempt to dequeue elements if they exist. A shared variable, named *filled*, records the number of elements in the queue. The code in the *get* function checks that the queue is nonempty (reading *filled* at line 10), decreases the count of elements in the queue (updating *filled* at line 20), and then dequeues the element.

The code has a concurrency bug because it does not ensure that the check and update of *filled* execute atomically. The lack of atomicity permits some unfavorable execution schedules in which the two consumer threads both attempt to dequeue the queue's last element. In that problematic case, both consumers read that the value of *filled* is 1, passing the test at line 10. One of the threads proceeds to decrement *filled* and dequeue the element. The other reaches the assertion at line 19, reads the value 0 for filled, and fails, terminating the execution. Figure 2(b) shows the interleaving of operations that leads to the failure in a concrete execution.

The next sections show how Symbiosis starts from a concrete failing execution (like the one in Figure 2(b)), computes a focused root cause, and produces a DSP to aid in debugging.

3.2. Symbolic Trace Collection

Like CLAP [Huang et al. 2013], Symbiosis avoids the overhead of directly recording the exact read-write linkages between shared variables that lead to a failure. Instead, Symbiosis collects only per-thread path profiles from a failing, concrete execution. As in prior work [Huang et al. 2013], Symbiosis's path profile for a thread consists of the sequence of executed basic blocks for that thread in the failing execution.

[1]By similar we mean that the alternate schedule comprises the same events as the failing schedule and adheres to the original execution path.

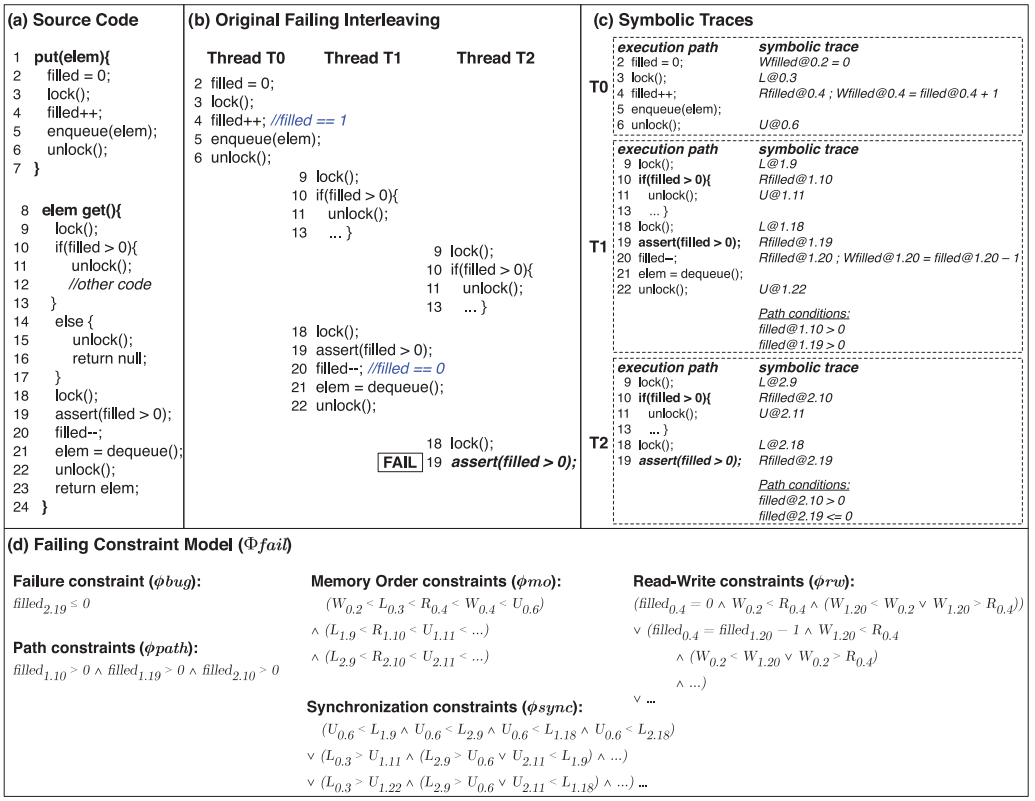

Fig. 2. Running example. (a) Source code. (b) Concrete failing execution. (c) Per-thread symbolic execution traces. (d) Failing Constraint Model.

Symbiosis uses the per-thread path profiles to guide a symbolic execution of each thread and to produce each thread's separate symbolic execution trace. Symbolic execution normally explores all paths, following the path along both branch outcomes. Symbiosis, in contrast, guides the symbolic execution to correspond to the per-thread path profiles by considering only paths that are compatible with the basic block sequence in the profile. As symbolic execution proceeds, Symbiosis records information about control-flow, failure manifestation, synchronization, and shared memory accesses in each per-thread symbolic execution trace. Together, the traces are compatible with the original, failing, multithreaded execution.

Each per-thread, symbolic, execution trace contains four kinds of information. First, each trace includes a path condition that permits the failure to occur. A trace's path condition is the sequence of control-flow decisions made during the trace's respective execution. Second, the trace for the thread that experienced the failure must include the event that failed (e.g., the failing assertion). Third, the trace must record synchronization operations, noting their type (e.g., lock, unlock, wait, notify, fork, join, etc.), and the synchronization variable involved (e.g., the lock address), if applicable. Fourth, the trace must record loads from and stores to shared memory locations. A key aspect of the shared memory access traces is that these are *symbolic*: Loads always read fresh symbolic values and stores may write either symbolic or concrete values. Recall from Section 2 that a symbolic value holds the last operation that manipulated a value. Also,

a symbolic value may, itself, be an expression that refers to other symbolic or concrete values.

Figure 2(c) illustrates a symbolic trace collection for our running example: It shows the execution path followed by each thread for the failing schedule in Figure 2(b) and the corresponding symbolic trace produced by Symbiosis. Each path condition in the trace represents a control-flow outcome in the original execution (e.g., *filled@2.10* > 0 denotes that thread T2 should read a value greater than zero from *filled* at line 10). Thread T2's trace includes the assertion that leads to the failure. Each trace includes both symbolic and concrete values in their memory access traces, as well as synchronization operations from the execution. Note that we slightly simplified the threads' traces to keep the figure uncluttered. *enqueue* and *dequeue* also access shared data but we only show operations that manipulate *filled* and perform synchronization because they are sufficient to illustrate the failure.

Symbiosis can leverage any technique for collecting concrete path profiles and generating symbolic traces. In our implementation of Symbiosis that targets C/C++, we use a technique very similar to the front-end of CLAP [Huang et al. 2013]: Symbiosis records a basic block trace and uses KLEE to generate per-thread symbolic traces conformant with the block sequence. Symbiosis for Java uses Soot [Vallée-Rai et al. 1999] to collect path profiles and JPF [Visser et al. 2004] for symbolic execution. The implementation details are described in Section 4.

With some additional engineering effort, Symbiosis could also use Pex [Tillmann and De Halleux 2008] for C# or general R&R techniques [Huang et al. 2010; Zhou et al. 2012].

3.3. Failing Schedule Generation

The symbolic, per-thread traces do not explicitly encode the multithreaded schedule that led to the failure. Symbiosis uses the information in the symbolic traces to construct a system of SMT constraints that encode information about the execution. The solution to those SMT constraints corresponds to a multithreaded schedule that ends in failure and is compatible with each per-thread symbolic trace. This section describes how the constraints are computed.

The SMT constraints refer to two kinds of unknown variables, namely the *value variables* for the fresh symbolic symbols returned by read operations and the *order variables* that represent the position of each operation from each trace in the final, multithreaded schedule. We notate value variables as $var_{t.l}$, meaning the value read from variable *var* by thread t at line l. We notate order variables as $Op_{t.l}$, meaning the order of instruction *Op* executed by thread t at line l, where *Op* can be a read (*R*), write (*W*), lock (*L*), unlock (*U*), or other synchronization operation such as wait/signal (our notation differs slightly from that of Huang et al. [2013] for clarity).

Figure 2(d) shows part of the system of SMT constraints generated by Symbiosis for our running example from the symbolic traces presented in Figure 2(c). The system, denoted Φ_{fail}, can be decomposed into five sets of constraints:

$$\Phi_{fail} = \phi_{path} \wedge \phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo},$$

where ϕ_{path} encodes the control-flow path executed by each thread, ϕ_{bug} encodes the occurrence of the failure, ϕ_{sync} encodes possible interthread interactions via synchronization, ϕ_{rw} encodes possible interthread interactions via shared memory, and ϕ_{mo} encodes possible operation reorderings permitted by the memory consistency model. The following paragraphs explain how Symbiosis derives each set of constraints from the symbolic execution traces.

Path Constraint (ϕ_{path})**.** The path SMT constraint encodes branch outcomes during symbolic execution. Symbiosis gathers path conditions by recording the branch outcomes along the basic block trace from the concrete path profile. A thread's path constraint is the conjunction of the path conditions for the execution of the thread in the symbolic trace. The ϕ_{path} constraint is the conjunction of all threads' path constraints. Each conjunct represents a single control-flow decision by constraining the value variables for one or more symbolic operands. In our running example, the shared variable *filled* is symbolic, resulting in a ϕ_{path} with three conjuncts. The three conjuncts express that the value of *filled* should be greater than 0 when thread T1 executes lines 10 and 19, as well as when thread T2 executes line 10. Figure 2(d) shows ϕ_{path} for our example.

Failure Constraint (ϕ_{bug})**.** The failure SMT constraint expresses the failure's necessary conditions. The constraint is an expression over value variables for symbolic values returned by some subset of read operations (e.g., those in the body of an *assert* statement). Figure 2(d) shows ϕ_{bug} for the running example, representing the sufficient condition for the assertion in thread T2 to fail.[2]

Synchronization Constraints (ϕ_{sync})**.** There are two types of synchronization SMT constraints: *partial order constraints* and *locking constraints*.

Partial order constraints represent the partial order of different threads' events resulting from *start*/*exit*/*fork*/*join*/*wait*/*signal* operations. Concretely, *start*, *join*, and *wait* operations are ordered with respect to *fork*, *exit*, and *signal* operations, respectively. The constraints for *start*/*fork* and *exit*/*join* are easy to model, as they exhibit a single mapping: The *start* event of a child thread must always occur after the corresponding *fork* operation in the parent thread, whereas the *exit* event of a child thread must always occur before the corresponding *join* operation in the parent thread. Let S_t represent the *start* event of a thread t, E_t the *exit* event of thread t, $F_{tp,tc}$ the *fork* operation of thread tc by thread tp, and $J_{tp,tc}$ the *join* operation of thread tc by thread tp. Their partial order constraints for these operations are then written as follows:

$$F_{tp,tc} < S_{tc}$$
$$E_{tc} < J_{tp,tc}.$$

The constraints for *wait* and *signal*, in contrast, are a little more complex. Similarly to CLAP [Huang et al. 2013], we use a binary variable that indicates whether a given *signal* operation is mapped to a *wait* operation. This is necessary because each *signal* operation can signal exactly one *wait* operation, although in theory it can be mapped to all *wait* operations on the same object. Let $\mathcal{SG}$ be the set with the *signal* operations sg on a given object and let $\mathcal{WT}$ be the set of *wait* operations wt on the same object but belonging to a thread different from that of sg. Also, let Sg and Wt denote the order of sg and wt, respectively, and b_{wt}^{sg} be the binary variable denoting whether sg is mapped to wt. The corresponding partial order constraints are the following:

$$\left(\bigvee_{\forall sg \in \mathcal{SG}} Sg < Wt \wedge b_{wt}^{sg} = 1 \right)$$
$$\bigwedge \sum_{wt \in \mathcal{WT}} b_{wt}^{sg} \leq 1.$$

The constraints above state that if a signal operation sg is mapped to a wait operation wt (i.e., $b_{wt}^{sg} = 1$), then sg must occur before wt and sg cannot signal any other *wait* operation rather than wt (i.e., $\sum_{wt \in \mathcal{WT}} b_{wt}^{sg} \leq 1$).

[2]For calls to external libraries, we also mark the result of the calls as symbolic (Section 4).

Locking constraints represent the mutual exclusion effects of *lock* (L) and *unlock* (U) operations. Let $\mathcal{P}$ denote the set of locking pairs on a given locking object and consider a particular pair L/U. The locking constraints entail the possible orderings between L/U and all the remaining pairs in $\mathcal{P}$ and are written as follows:

$$\bigwedge_{\forall L'/U' \in \mathcal{P}} U < L' \vee \bigvee_{\forall L'/U' \in \mathcal{P}} \left(U' < L \wedge \bigwedge_{\forall L''/U'' \in \mathcal{P},\, L''/U'' \neq L'/U'} U < L'' \vee U'' < L' \right).$$

The constraint above is a disjunction of SMT expressions representing two possible cases. In the first case, L/U is the first pair acquiring the lock and, therefore, U happens before L'. In the second case, L/U acquires the lock released by another pair L'/U', hence U' happens before L. Moreover, for any other pair L''/U'', either L'' acquires the lock after U or L' acquires the lock released by U''. Figure 2(d) shows a subset of the locking constraints for our running example that involve the *lock*/*unlock* pair of thread T0 ($L_{0:3}$, $U_{0:6}$).

Read-Write Constraints (ϕ_{rw})**.** Read-write SMT constraints encode the matching between read and write operations that leads to a particular read operation reading the value written by a particular write operation. Read-write constraints model the possible interthread interactions via shared memory. A read-write constraint encodes that, for every read operation r on a shared variable v, if r is matched to a write w of the same variable, then the order variable (and hence execution order) for all other writes on v are either less than that of w or greater than that of r. The constraint also implies that r's value variable takes on the symbolic value written by w. Note that read-write SMT constraints are special in that they affect order variables and value variables.

Let r_v be the value returned by a read r on v, and let W be a set of writes on v. Using R to denote the order of r and W_i the order of write w_i in $\mathcal{W}$, ϕ_{rw} can be written as follows:

$$\bigvee_{\forall w_i \in \mathcal{W}} \left(r_v = w_i \wedge W_i < R \bigwedge_{\forall w_j \in \mathcal{W}, w_j \neq w_i} W_j < W_i \vee W_j > R \right).$$

For example, in our running example, thread T0 reads *filled* at line 4. If T0 reads 0 at that point, then the most recent write to *filled* must be the one at line 2. The matching of that read and write implies that the order of the write must precede read operation (i.e., $W_{0.2} < R_{0.4}$), and that all the other writes to *filled* (e.g., $W_{1.20}$) either occur before $W_{0.2}$ or after $R_{0.4}$. The same reasoning is also applied for the remaining reads of the program on symbolic variables.

Memory Order Constraints (ϕ_{mo}). The memory-order constraints specify the order in which instructions are executed in a specific thread. Although is possible to express different memory consistency models [Huang et al. 2013], in this article we opted not to focus on relaxed memory ordering, instead focusing on subschedule generation and differential schedule projections. Therefore, here we consider sequential consistency (SC) only, meaning statements in a thread execute in program order. For the running example in Figure 2(b), the memory order constraint requires that operations in thread T0 respect the constraint $W_{0.2} < L_{0.3} < R_{0.4} < W_{0.5} < U_{0.6}$.

3.4. Root Cause Subschedule Generation

Each order variable referred to by an SMT constraint represents the ordering of two program events from the separate single-threaded symbolic traces. A binding of truth values to the SMT order variables corresponds directly to an ordering of operations in the otherwise unordered, separate, per-thread traces. Solving the constraint system binds truth values to variables, producing a multithreaded schedule. The constraint system includes a constraint representing the occurrence of the failure, so the produced multithreaded schedule manifests the failure (ϕ_{bug}). Solving the generated SMT formulae, Symbiosis produces a full, failing, multithreaded schedule ϕ_{fsch}. The entire multithreaded schedule may be long and complex and may contain information that is irrelevant to the root cause of the failure. Symbiosis uses a special SMT formulation to produce a *root-cause subschedule* that prunes some operations in the full schedule but preserves event orderings that are *necessary* for the failure to occur. To compute the root-cause subschedule, Symbiosis generates a new constraint system, denoted Φ_{root}, that is *designed to be unsatisfiable* in a way that reveals the necessary orderings. Symbiosis leverages the ability of the SMT solver to produce an explanation of why a formula was unsatisfiable to report only those necessary orderings.

To build the root-cause subschedule SMT formula, Symbiosis logically inverts the *failure constraint*, effectively requiring the failure not to occur (i.e., $\neg\phi_{bug}$). Symbiosis adds constraints to the formula that directly encode the event orders in ϕ_{fsch} (i.e., the full, failing schedule that was previously computed). The complete root-cause subschedule formula is then written as follows:

$$\Phi_{root} = \phi_{path} \wedge \neg\phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{fsch}.$$

The original SMT formula that Symbiosis used to find the full failing schedule considers *all* possible multithreaded schedules that are consistent with the symbolic, per-thread schedules. In contrast, the root-cause subschedule SMT formula adds the failing schedule ϕ_{fsch} constraint, accommodating *only* the full, failing schedule. Combining the inverted failure constraint and the ordering constraints for the full, failing schedule makes an unsatisfiable constraint formula: The inverted failure constraint requires the failure not to occur and the failing schedule's ordering constraints require the failure to occur.

When an SMT solver, like Z3, attempts to solve the unsatisfiable formula, it produces an unsatisfiable (UNSAT) core which is a subset of constraint clauses that conflict, leaving the formula unsatisfiable. The UNSAT core for Φ_{root} encodes the subset of clauses that conflict because the ϕ_{fsch} requires the failure to occur and $\neg\phi_{bug}$ requires the failure not to occur. The event orderings that correspond to those conflicting constraints are the ones in ϕ_{fsch} that imply ϕ_{bug}. Those orderings are a necessary condition for the failure; their corresponding constraints, together with $\neg\phi_{bug}$, are responsible for the unsatisfiability of Φ_{root}. Reporting the subschedule corresponding to the UNSAT core yields fewer total events than are in the full, failing schedule, yet includes event orderings necessary for the failure.

Figure 3(a) shows a possible failing schedule produced by the constraint system corresponding to the execution depicted in Figure 2(d). The failure constraint ϕ_{bug} requires the corresponding execution to manifest the failure. The generated path and memory access constraints are compatible with the failure and the system is satisfiable, producing the failing execution trace shown (ϕ_{fsch}). Note that Symbiosis inserts a *synthetic unlock* event *($U_{2.20}$)* in the model, in order to preserve the correct semantics of synchronization constraints (see Section 4).

In Figure 3(b), the failure constraint is *negated*, requiring the corresponding execution not to manifest the failure (i.e., $filled_{2.19} > 0$ and the assertion at line 19 does not fail). On the other hand, ϕ_{fsch} satisfies only the dataflows encoded in ϕ_{rw} that correspond

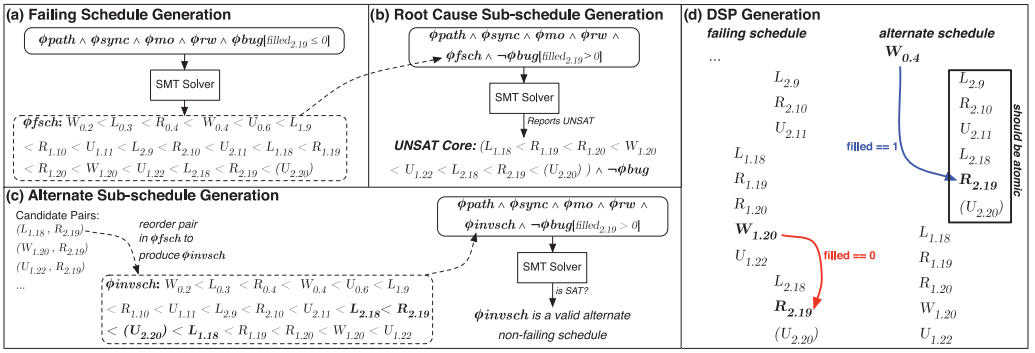

Fig. 3. Root cause and alternate schedule generation. (a) Possible failing schedule produced by the SMT solver for the constraint system in Figure 2(d) ($(U_{2.20})$ represents a synthetic unlock event). (b) Root cause subschedule, which corresponds to the UNSAT core produced by the solver. (c) Candidate pair reordering and respective alternate schedule. (d) Differential Schedule Projection generated by Symbiosis.

to the failing schedule, which means that $R_{2.19}$ is forced to receive the value written by $W_{1.20}$. Consequently, $filled_{2.19}$ becomes 0 instead of greater than 0, as required by the negated failure constraint (note that $R_{2.19}$ defines the value of $filled_{2.19}$ read by the assertion). Since both subformulae ϕ_{fsch} and $\neg\phi_{bug}$ conflict with each other, the solver yields unsatisfiable for this model. In addition, the solver outputs the UNSAT core containing the subset of constraints of ϕ_{fsch} that conflict with $\neg\phi_{bug}$ (see Figure 3(b)). Note that the UNSAT core shows why Φ_{root} is unsatisfiable: The negated failure constraint conflicts with the subset of ordering constraints from ϕ_{fsch} that cause *filled* to be less than 0 when thread 2 executes line 19.

In our experience, the UNSAT core produced by Z3 is typically not minimal (although it is always an overapproximation of the root cause). As a result, while helpful, an UNSAT core alone is not sufficient for debugging and necessitates a differential schedule projection to isolate a bug's root cause.

3.5. Alternate Subschedule Generation

In addition to reporting the bug's root cause, Symbiosis also produces *alternate, nonfailing subschedules*. These alternate subschedules are *nonfailing* variants of the root-cause subschedule, with the order of a single pair of events reversed. Alternate subschedules are the key to building *differential schedule projections* (Section 3.6). Symbiosis generates alternate, nonfailing subschedules after it identifies the root cause. To generate an alternate subschedule, Symbiosis selects a pair of events from different threads that were included in the bug's root cause. Symbiosis then generates a new constraint model, like the one used to identify the root cause. We call this model Φ_{alt}. The Φ_{alt} model includes the inverted failure constraint. The model also includes a set of constraints, denoted ϕ_{invsch}, that encode the original full, failing schedule, *except the constraint representing the order of the selected pair of events is inverted*. Inverting the order constraint for the pair of events yields the following new constraint model:

$$\Phi_{alt} = \phi_{path} \wedge \neg\phi_{bug} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{invsch}.$$

The new Φ_{alt} model corresponds to a different, full execution schedule in which the events in the pair occur in the order opposite to that in the full, failing schedule. If this new model is satisfiable, then reordering the pair of events in the full, failing schedule produces a new alternate schedule in which the failure does not manifest, as shown in Figure 3(c).

If there are many event pairs in the root cause, then Symbiosis must generate and attempt to solve many constraint formulae. Symbiosis systematically evaluates a set of candidate pairs in a fixed order, choosing the pair separated by the fewest events in the original schedule first. The reasoning behind this design choice is that events in a pair that are further apart are less likely to be meaningfully related and, thus, less likely to change the failure behavior when their order is inverted. The experimental results in Section 5.3 show that this heuristic is effective for most cases.

By default, we configured Symbiosis to stop after finding a single alternate, nonfailing schedule. However, the programmer can instruct Symbiosis to continue generating alternate schedules, given that studying sets of schedules may reveal useful invariants [Lucia et al. 2011].

Arbitrary operation reorderings may yield infeasible schedules. Reordering may change interthread data flow, producing values that are inconsistent with a prior branch dependent on those values. The inconsistency between the data and the execution path makes the execution infeasible. Symbiosis produces only feasible schedules by including path constraints in its SMT model. If a reordering leads to inconsistency, then the SMT path constraints become unsatisfiable and Symbiosis produces no schedule. We denote this property as *feasibility*.

Moreover, our event pair reordering technique has the property of guaranteeing that if there exists a feasible alternate schedule that adheres to the original execution path and prevents the failure, then Symbiosis finds it. We denote this property as *1-Recall*.[3]

The feasibility and 1-Recall properties are proved in the following paragraphs.

3.5.1. Alternate Schedule Feasibility. Intuitively, what we want to show is that any alternate schedule, resulting from a reordering of events in a failing schedule, is feasible if deemed as satisfiable by Symbiosis's solver. To formally define the feasibility of a schedule, we rely on the concept of (sequential) consistency proposed in Herlihy and Wing [1990] and adapt the notation used in Huang et al. [2014]. Hence, a schedule is considered feasible if it is sequentially consistent.

Consider a *schedule* σ to be a sequence of *events* e. Events are operations performed by threads on *concurrent objects* (e.g., shared variables, locks, etc.) for data sharing and synchronization purposes. A concurrent object is behaviorally defined by a set of atomic operations and by a serial specification of its legal computations, when performed in isolation [Herlihy and Wing 1990]. For instance, a shared variable is a concurrent object containing read and write operations, whose serial specification states that each read returns the value of the most recent write.

Let σ_e denote the prefix of schedule σ up to e (inclusive): If $\sigma = \sigma_1 e \sigma_2$, then σ_e is $\sigma_1 e$. Moreover, let $\sigma_{[op,var,thread]}$ represent the restriction of σ to events involving operations of type *op*, on variable *var*, by thread *thread*. For instance, $\sigma_{[R,*,1]}$ represents the projection of schedule σ to read operations performed by thread 1 on all shared variables; $\sigma_{[W,v,*]}$ consists of the restriction of σ to write operations on shared variable v by any thread, and so on.

An alternate schedule of σ is a schedule σ' that exhibits the same per-thread schedules as σ but permits different orders of events from different threads: $\sigma'_{[*,*,t]} = \sigma_{[*,*,t]}$, for each thread t.

[3]Similarly to the definition in the field of information retrieval, we use *recall* to denote the fraction of relevant solutions (i.e., solutions that prevent the failure and adhere to the same path constraints as the failing schedule) that Symbiosis successfully outputs. The *1-Recall* property, thus, indicates that all solutions output by Symbiosis are relevant.

Schedule σ is *(sequentially) consistent* iff $\sigma_{[*,o,*]}$ satisfies o's serial specification for any concurrent object o [Herlihy and Wing 1990]. More formally, a schedule σ is sequentially consistent when it meets the following requirements:

Read Consistency. A read event returns the value written by the most recent write event on the same variable. Formally, if e is a read event in σ on variable v, then $data(e) = data(last_{write}(\sigma_{e[W,v,*]}))$, where $data(e)$ gives the value returned by the read event e, and $data(last_{write}(\sigma_{e[W,v,*]}))$ is the last value written on variable v in σ_e.

Lock Mutual Exclusion. Each *unlock* (U) event is preceded by a *lock* (L) event on the same lock object by the same thread, and a locking pair cannot be interleaved by any other L or U event on the same object. Formally, for any lock object l, if $\sigma_{[*,l,*]} = e_1 e_2 ... e_n$ then $e_k = L$ for all odd indexes $k \leq n$, $e_k = U$ for all even indexes $k \leq n$, and $thread(e_k) = thread(e_{k+1})$ for all odd indexes k with $k < n$.

Must Happen-Before. Let $start(t)$/$exit(t)$ be the first/last event of thread t. Then a *start* event in a thread t' can only occur in σ after t' is forked by thread t, that is, for any event $e = start(t')$ in σ, $\sigma_{[*,*,t']}$ begins with e and there exists precisely one $fork(t, t')$ event in σ_e. Similarly, a *join* event in a thread t can only occur in σ after t' has ended, that is, for any event $e = exit(t')$ in σ, $\sigma_{[*,*,t']}$ terminates with e and there exists precisely one $join(t, t')$ event in σ_e.

Note that branch conditions do not have serial specifications, and hence they can affect the control flow of an execution but not the consistency of its schedule. However, since we do not have information regarding operations in other execution paths rather than the one captured in the concrete trace, we conservatively assume that an alternate schedule must have the same control flow as the failing schedule (except for the assertion corresponding to the bug condition). Therefore, we can say that an alternate schedule is feasible iff it meets the aforementioned consistency requirements and adheres to the branch conditions of the failing schedule. We now prove the following theorem:

THEOREM 3.1 (FEASIBILITY). *Given a feasible failing schedule σ, any alternate schedule σ' that is satisfiable by Symbiosis's solver is feasible.*

PROOF. To prove the theorem above, we will first show that any alternate schedule that satisfies our SMT constraint model in Section 3.5 is sequentially consistent, that is it provides *read consistency*, *lock mutual exclusion*, and *must happen-before* properties. Then we will show that any alternate schedule that is considered satisfiable by Symbiosis's solver also satisfies the same path conditions as the failing schedule.

The *read consistency* property requires that a read event returns the value written by the most recent write event on the same variable. In our constraint model, this property holds from the read-write constraints. As shown in Section 3.3, the read-write constraints encode all possible linkages between reads and writes in the symbolic traces. This means that, for a given read event e on a shared variable v, the read-write formulae include a disjunction of constraints encoding the mapping between the value returned by e and all the existing writes on v. Hence, considering r_v to be the value returned by e in a schedule σ_e (i.e., $r_v = data(e)$), there always exists a write w_l such that $w_l = data(last_{write}(\sigma_{e[W,v,*]}))$. In other words, w_l is the most recent write on v with respect to e and satisfies the following constraint: $r_v = w_l \wedge W_l < R_e \bigwedge_{\forall w_j \in \mathcal{W}, w_j \neq w_l} W_j < W_l \vee W_j > R_e$, where capital letters signify order variables, that is, W_l and R indicate the order of write w_l and read event e in schedule σ, respectively.

The *lock mutual exclusion* property holds from the locking constraints in our SMT model, as they encode the mutual exclusion effects of the acquisition and release of lock objects. To prove this, we show that any locking order that satisfies our locking

constraint formulae is of form $L_1U_1L_2U_2L_kU_k \ldots L_nU_n, \forall_{k \leq n}$, with $thread(L_k) = thread(U_k)$, as required by the lock mutual exclusion property. Recall the locking constraints for a locking pair L/U on a lock object l in a thread t[4], as shown in Section 3.3:

$$i) \bigwedge_{\forall L'/U' \in \mathcal{P}} U < L' \vee$$

$$ii) \bigvee_{\forall L'/U' \in \mathcal{P}} \left(U' < L \wedge \bigwedge_{\forall L''/U'' \in \mathcal{P}, L''/U'' \neq L'/U'} U < L'' \vee U'' < L' \right).$$

According to our SMT constraint system, it must be the case that either (i) L/U is the first locking pair in the schedule or (ii) it acquires the lock on l released by a previous locking pair L'/U' (and, here, all the other locking pairs either occur before L'/U' or after L/U). Let k denote the order in which the pair L/U holds the lock in a given schedule, that is, L_k/U_k is the k^{th} pair acquiring the lock in the schedule. If $k = 1$, then L_k/U_k is the first pair acquiring the lock on l (i.e., L_1/U_1), which means that the portion of the locking constraint formula for L_1/U_1 that will be true is (i): $U_1 < L_2 \wedge U_1 < L_3 \wedge \ldots \wedge U_1 < L_n$.

In turn, when $1 < k \leq n$, the pair L_k/U_k will acquire the lock released by the pair L_{k-1}/U_{k-1}, thus satisfying the constraint subformula (ii) instead: $U_{k-1} < L_k \bigwedge_{\forall 1 \leq j \leq n, j \neq k, j \neq k-1}, U_k < L_j \vee U_j < L_{k-1}$. However, note that, when $k = n$, the constraint will be of type: $U_1 < L_n \wedge U_2 < L_n \wedge \ldots \wedge U_{n-1} < L_n$, meaning the pair L_n/U_n is the last one acquiring the lock.

In sum, since the constraints enforce that a given pair L_k/U_k can only acquire the lock released by a single pair L_{k-1}/U_{k-1}, it follows that each locking pair will have a unique value of k, that is, a unique position in the schedule. Therefore, any global locking order that satisfies our locking constraints is of the form $L_1U_1L_2U_2L_kU_k \ldots L_nU_n, \forall_{k \leq n}$.

The *must happen-before* property follows directly from the partial order constraints described in Section 3.3: The operations S_t, E_t, $F_{tp,tc}$, $J_{tp,tc}$ correspond, respectively, to the events $start(t)$, $exit(t)$, $fork(t, t')$, and $join(t, t')$ mentioned in the beginning of this section. Therefore, the partial order constraints in our SMT model directly encode the necessary happen-before guarantees required for a schedule to be sequentially consistent according to Herlihy and Wing [1990]. □

3.5.2. Alternate Schedule 1-Recall. In addition to feasibility, another important property that we are interested in proving is the *1-Recall* property of Symbiosis with respect to finding alternate, nonfailing schedules via event pair reordering. The 1-Recall property can be defined as the following theorem:

THEOREM 3.2 (1-RECALL). *Given a failing schedule σ, let $\mathcal{A}$ be the set of feasible alternate schedules of σ that adhere to the same execution path and prevent the failure. Let $\mathcal{S}$ be the set of alternate schedules output*[5] *by Symbiosis. If $|\mathcal{A}| \geq 1$, then $|\mathcal{S}| \geq 1 \wedge \mathcal{S} \subseteq \mathcal{A}$.*

PROOF. Informally, the above theorem states that, given a feasible failing schedule σ, if there exists a feasible (nonfailing) alternate schedule σ', then Symbiosis will find it.

We divide the proof of the theorem into three steps. First, we define the condition necessary and sufficient that any alternate schedule $\sigma_a \in \mathcal{A}$ must verify in order not to

[4]Note that our locking constraints operate over the locking pairs extracted from each thread's symbolic traces, therefore it is always the case that $thread(L) = thread(U)$ for every pair L/U.

[5]We say that an alternate schedule is output by Symbiosis iff it is deemed as satisfiable by the solver.

trigger the failure. Second, we show that any alternate schedule σ' output by Symbiosis meets this condition (i.e., $\mathcal{S} \subseteq \mathcal{A}$). Third, we show that if there exist feasible alternate schedules that prevent the failure, then Symbiosis finds at least one of them (i.e., $|\mathcal{A}| \geq 1 \Rightarrow |\mathcal{S}| \geq 1$).

For a given failing execution, let $\mathcal{F}$ be the *minimal* set of events that are sufficient to trigger the concurrency failure, $\sigma_{[\mathcal{F}]}$ be the projection of the failing schedule σ to the events in $\mathcal{F}$ (i.e., $\sigma_{[\mathcal{F}]}$ is the minimal ordered sequence of events that causes the concurrency failure), and $\sigma_{a[\mathcal{F}]}$ be the projection of the alternate schedule $\sigma_a \in \mathcal{A}$ to the events in $\mathcal{F}$.

It has been shown that if $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$, then σ_a does not trigger the failure [Lucia and Ceze 2013]. This result has been named *Avoidance-Testing Duality* and, informally, states that, given *any* ordered sequence of events that trigger a concurrency failure, it suffices to perturb just *one* pair of events to avoid the failure [Lucia and Ceze 2013].

Since, for any alternate schedule $\sigma_a \in \mathcal{A}$, $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ must hold, to prove that $\mathcal{S} \subseteq \mathcal{A}$, using the result above, we only need to show that $\forall_{\sigma' \in \mathcal{S}}, \sigma'_{[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$.

Consider $\mathcal{R}$ to be the set of events belonging to the root cause produced by the constraint formula Φ_{root} (see Section 3.4). Recall that Symbiosis generates alternate schedules by (exhaustively) selecting pairs of events from $\mathcal{R}$ to be inverted in the original failing schedule.

Let $\sigma' = \mathit{invert}(\sigma, e_j, e_k)$ be the alternate schedule σ' that Symbiosis produces by reordering the j^{th} and k^{th} events in σ, with $j < k$. Considering t_k as the thread of event e_k (i.e., $t_k = \mathit{thread}(e_k)$) and rewriting $\sigma = e_1 e_2 \dots e_{j-1} e_j e_{j+1} \dots e_{k-1} e_k e_{k+1} \dots e_n$ as $\sigma = \alpha e_j \beta e_k \gamma$, then $\sigma' = \mathit{invert}(\sigma, e_j, e_k) = \alpha \beta_{[*,*,t_k]} e_k e_j \beta \setminus \beta_{[*,*,t_k]} \gamma$. Here, $\beta_{[*,*,t_k]}$ corresponds to the events by thread t_k that occur between e_j and e_k in σ, and $\beta \setminus \beta_{[*,*,t_k]}$ corresponds to the set of events β excluding the events in $\beta_{[*,*,t_k]}$. In other words, the alternate schedule σ' is computed by placing e_k right before e_j, as well as all the events, belonging to the same thread of e_k, that occur between e_j and e_k in the failing schedule σ.

Let now e_{f_j}, e_{f_k} be a pair of events selected by Symbiosis to be inverted, such that $e_{f_j}, e_{f_k} \in \mathcal{F}$. Note that we know that $\exists_{e_j, e_k \in \mathcal{R}} : e_j, e_k \in \mathcal{F}$ because the UNSAT core output by the solver (which corresponds to $\mathcal{R}$) always contains, at least, the events belonging to the minimal sequence of events that leads to the bug. Therefore, $\mathcal{F} \subseteq \mathcal{R}$ and $\exists_{e_j, e_k \in \mathcal{R}} : e_j, e_k \in \mathcal{F}$ holds by construction.

If $\sigma_{[\mathcal{F}]} = \alpha_f e_{f_j} \beta_f e_{f_k} \gamma_f$ is the minimal ordered sequence of events that causes σ to fail, and $\sigma' = \mathit{invert}(\sigma, e_{f_j}, e_{f_k})$ is the alternate schedule output by Symbiosis, then $\sigma'_{[\mathcal{F}]} = \alpha_f \beta_{f[*,*,\mathit{thread}(e_{f_k})]} e_{f_k} e_{f_j} \beta_f \setminus \beta_{f[*,*,\mathit{thread}(e_{f_k})]} \gamma_f$. Thus, $\sigma'_{[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ is true and $\sigma' \in \mathcal{A}$.

On the other hand, note that, for all event pairs $e_j, e_k \in \mathcal{R}$ and $\sigma'' = \mathit{invert}(\sigma, e_j, e_k)$, if $e_j, e_k \notin \mathcal{F}$, then $\sigma''_{[\mathcal{F}]} = \sigma_{[\mathcal{F}]}$ will hold and the solver will yield *unsatisfiable*. Thus, $\sigma'' \notin \mathcal{S}$ and it is not output by Symbiosis.

Finally, we prove that $|\mathcal{A}| \geq 1 \Rightarrow |\mathcal{S}| \geq 1$ by contradiction. Suppose $|\mathcal{A}| \geq 1$ and $\mathcal{S} = \emptyset$, then it must be the case that there is a feasible nonfailing, alternate schedule σ_a that verifies the condition $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ and is not obtainable by reordering pairs of events in $\mathcal{R}$ (given that Symbiosis attempts to invert all pairs of events in $\mathcal{R}$). However, if σ_a is not the result of a reordering of events in $\mathcal{R}$, then it does not comprise events from $\mathcal{F}$ (since $\mathcal{F} \subseteq \mathcal{R}$). Consequently, σ_a cannot belong to $\mathcal{A}$, because it does not include the same set of events nor adheres to the same execution path as the original failing schedule σ, as required by the definition of $\mathcal{A}$. This contradicts our assumption. □

As final remark, note that Symbiosis requires the alternate, nonfailing schedule to adhere to the same control-flow as the original failing schedule. This means that, for concurrency bugs whose root cause is related to schedule-sensitive branches [Huang and Rauchwerger 2015] (i.e., path- and schedule-dependent bugs), Symbiosis is not

able to produce an alternate schedule. We leave the support for isolating path- and schedule-dependent bugs for future work.

3.6. Differential Schedule Projection Generation

DSP is a novel debugging methodology that uses root-cause subschedules and nonfailing alternate subschedules. The key idea behind debugging with a DSP is to show the programmer the salient differences between failing, root-cause schedules and nonfailing, alternate schedules. Examining those differences helps the programmer understand how to *fix* the bug, rather than to help them understand the failure only, like techniques that solely report failing schedules.

Concretely, a DSP consists of a pair of subschedules decorated with several pieces of additional information. The first subschedule is the root-cause subschedule, which is the *source* of the projection. The second subschedule is an excerpt from the alternate, nonfailing schedule, which is the *target* of the projection.

The order of memory operations differs between the schedules and, as a result, the outcome of some memory operations may differ. A read may observe a different write's result in one schedule than it observed in another, or two writes may update memory in a different order in one schedule than in another, leaving memory in a different final state. These differences are precisely the changes in dataflow that contribute to the failure's occurrence. Symbiosis highlights the differences by reporting *dataflow variations*: dataflow between operations in the source subschedule that do not occur in the target subschedule and *vice versa*.

To simplify its output, Symbiosis reports only a subset of operations in the source and target subschedules. An operation is included if it makes up a dataflow variation or if it is one of a pair of operations that occur in a different order in one subschedule than in the other. Alternate, nonfailing schedules vary in the order of a single pair of operations, so all operations that precede both operations in the pair occur in the same order in the source and target subschedules. Symbiosis does not report operations in the common prefix, unless it is involved in a dataflow variation. By selectively including only operations related to dataflow and ordering differences, a DSP focuses programmer attention on the changes to a failing execution that lead to a nonfailing execution. Understanding those changes are the key to changing the program's code to fix the bug. For instance, the DSP in Figure 3(d) shows that the dataflow $W_{1.20} \rightarrow R_{2.19}$ (in ϕ_{fsch}) changes to $W_{0.4} \rightarrow R_{2.19}$ (in ϕ_{invsch}). This dataflow variation is the failure's root cause. In addition, note that, by reordering the events, the DSP also suggests that the block of operations $L_{2.9}$–$(U_{2.20})$ should execute atomically, which indeed fixes the bug.

3.7. DSP Optimization: Context Switch Reduction

The SMT solver does not take into account the number of context switches when solving the failing constraint system. As a consequence, the failing schedule produced may exhibit a fine-grained entanglement of thread operations, which hinders analysis. Although DSPs help obviate most of the interleaving complexity by pruning the common portions between the failing and the alternate schedules, they may still depict unnecessary data dependencies. Figure 4(a) illustrates this scenario.

The program in the figure contains two threads (*T1* and *T2*), three shared variables (*x*, *y*, and *z*), and an assertion that checks whether *x!=0*. The DSP in Figure 4(a) shows a possible failing schedule and depicts the dataflow variations with respect to the corresponding alternate schedule. We can see that the DSP highlights differences in the dataflow for shared variables *x*, *y*, and *z*, although solely the one for *x* is indeed related to the bug's root cause. Note that, for this example, the alternate schedule is produced by inverting the event *x=0* (in *T2*) with *assert(x!=0)* (in *T1*).

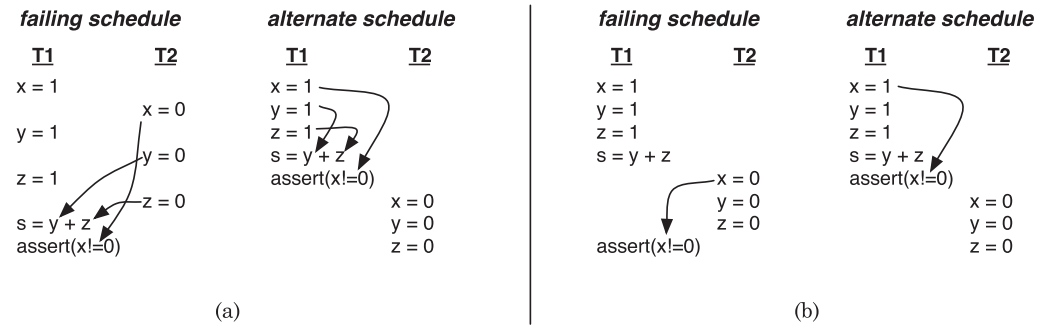

Fig. 4. Context Switch Reduction. (a) DSP *without* context switch reduction; (b) DSP *with* context switch reduction. Arrows depict execution dataflows.

To mitigate the amount of irrelevant data dependencies in DSPs, we apply a preprocessing stage to reduce the number of context switches in the failing schedule prior to the generation of the alternate schedule. Since finding the minimal number of context switches that triggers a failure is NP-hard [Jalbert and Sen 2010], we developed an algorithm inspired in Tinertia [Jalbert and Sen 2010], which is a trace simplification heuristic that runs in polynomial time with respect to the size of the trace. Our context switch reduction (CSR) algorithm is described in Algorithm 1. We study the impact of CSR in DSP generation in Section 5.3.3.

ALGORITHM 1: Context Switch Reduction (CSR)

Input: failing schedule σ; constraint system Φ_{fail}
Output: failing schedule σ_{cur} with the number of context switches (potentially) reduced

```
1  σ_cur ← σ
2  repeat
3      σ_old ← σ_cur
4      for i = 1 to |σ_cur| do
5          σ_tmp ← moveUpSeg(σ_cur, i)
6          if solve(Φ_fail, σ_tmp) is satisfiable then
7              σ_cur ← σ_tmp
8          end
9      end
10     for i = |σ_cur| to 1 do
11         σ_tmp ← moveDownSeg(σ_cur, i)
12         if solve(Φ_fail, σ_tmp) is satisfiable then
13             σ_cur ← σ_tmp
14         end
15     end
16 until numCS(σ_old) ≤ numCS(σ_cur);
17 return σ_cur
```

The CSR algorithm receives a failing schedule σ and the constraint system Φ_{fail} (see Section 3.3) as input. All the context switch reduction actions are applied to σ_{cur}, and whenever the schedule σ_{tmp}, resulting from the application of an action, satisfies the constraint system Φ_{fail}, that schedule is stored into σ_{cur}.

CSR starts by initializing σ_{cur} to the input schedule σ and then enters the main loop (lines 2–16). In lines 4–9, the algorithm does a forward pass over the schedule and applies the *moveUpSeg* action for each event e in the schedule (line 5). This action

operates at the *thread segment*[6] level; therefore, it has an effect only if e is the last operation of the segment (otherwise it just proceeds to the next event). When e is in the tail of the segment, *moveUpSeg* finds the next thread segment after e that is executed by the same thread and merges it with the thread segment that contains e. If this move produces a schedule that still satisfies the constraint model (line 6), then we have successfully found a schedule with one context switch less and store it into σ_{cur} (line 7). As an example of this action, consider the event *x = 1* in the failing schedule of Figure 4(a). If we apply *moveUpSeg* to this event, then its thread segment will be augmented with the next thread segment by the same thread (i.e., *y = 1*). The schedule resulting from this move will then be *x = 1; y = 1; x = 0; (. . .)*.

The next step of the CSR algorithm does a backwards pass over the schedule and applies the *moveDownSeg* action for each event e in the schedule (lines 10–15). Symmetrically to *moveUpSeg*, *moveDownSeg* looks for the previous thread segment before e that is executed by the same thread and merges it with the thread segment that contains e. Hence, *moveDownSeg* only has an effect if e is the first action of the thread segment. This technique is particularly helpful to eliminate context switches in the presence of partial order invariants. For instance, consider two thread segments A and B of thread *T1*, interleaved by a segment C of thread *T2*. If B contains a *join* event and C contains an *exit* event, then *moveUpSeg* will yield unsatisfiable when pulling B upwards to be merged with A, because B cannot occur before C. In contrast, *moveDownSeg* will be valid because A can be merged down with B and execute after C without breaking the partial order invariant.

At the end of each iteration of the main loop, CSR computes the number of context switches of σ_{cur} (given by $numCS(\sigma_{cur})$) and compares this value to that of σ_{old}. If σ_{cur} contains fewer context switches than σ_{old}, then the algorithm proceeds to further simplify σ_{cur}. Otherwise, CSR terminates and returns σ_{cur} as the simplified schedule.

Figure 4(b) shows the DSP obtained after applying the CSR algorithm to the failing schedule in Figure 4(a). We can see that the single dataflow variation that is now depicted is the one that explains the failure.

It should be noted that Symbiosis could also leverage other trace simplification techniques, which do not rely on SMT solver invocations. For example, SimTrace [Huang and Zhang 2011] is a static technique that reduces the number of context switches in an execution schedule by computing a graph of dependences of the events in the schedule.

4. IMPLEMENTATION

4.1. Instrumenting Compiler and Runtime

Our Symbiosis prototype implements trace collection for both C/C++ and Java programs. C/C++ programs are instrumented via an LLVM function pass. Java programs are instrumented using Soot [Vallée-Rai et al. 1999], which injects path logging calls into the program's bytecode. Like CLAP, we assign every basic block with a static identifier and, at the beginning of each block, we insert a call to a function that updates the executing thread's path. The function logs each block as the tuple *(thread Id, basic block Id)* whenever the block executes. The path logging function is implemented in a custom library that we link into the program. Although our prototype is fully functional, it has not been fully optimized yet. For instance, lightweight software approaches (e.g., Ball and Larus [1994]) or a hardware accelerated approaches (e.g., Vaswani et al. [2005]) could also be used to improve the efficiency of path logging. The Symbiosis prototype is publicly available at https://github.com/nunomachado/symbiosis.

[6]We consider a *thread segment* to be a maximal sequence of consecutive events by the same thread.

4.2. Symbolic Execution and Constraint Generation

Symbiosis's guided symbolic execution for C/C++ programs has been implemented on top of KLEE [Cadar et al. 2008]. Since KLEE does not support multithreaded executions, similarly to CLAP, we fork a new instance of KLEE's execution to handle each new thread created. We also disabled the part of KLEE that solves path conditions to produce test inputs because Symbiosis does not use them. For Java programs, we have used Java PathFinder (JPF) [Visser et al. 2004]. In this case, we have disabled the handlers for *join* and *wait* operations to allow threads to proceed their symbolic execution independently, regardless of the interleaving. Otherwise, we would have to explore different possible thread interleavings when accessing these operations, in order to find one conforming with the original execution.

Additionally, we made the following changes to both symbolic execution engines. First, we ignore states that do not conform with the threads' path profiles traced at runtime, which allows to guide the symbolic execution along the original paths alone. Second, we generate and output a per-thread symbolic trace containing read/write accesses to shared variables, synchronization operations, and path conditions observed across each execution path.

Consistent Thread Identification. Symbiosis must ensure threads are consistently named between the original failing execution and the symbolic executions. We use a technique previously used in jRapture [Steven et al. 2000] that relies on the observation that each thread spawns its children threads in the same order, regardless of the global order among all threads. Symbiosis instruments thread creation points, replacing the original PThreads/Java thread identifiers with new identifiers based on the parent-children order relationship. For instance, if a thread t_i forks its jth child thread, the child thread's identifier will be $t_{i:j}$.

Marking Shared Variables as Symbolic. Precisely identifying accesses to shared data, in order to mark shared variables as symbolic, is a difficult program analysis problem, which is orthogonal to our work. Although it is possible to conservatively mark all variables as symbolic, varying the number of symbolic variables varies the size and complexity of the constraint system. For C/C++ programs, we manually marked shared variables as symbolic, like prior work [Huang et al. 2013]. We also marked variables symbolic if their values were the result of calls to external libraries not supported by KLEE. For Java programs, we use Soot's *thread-local objects* (TLO) static escape analysis strategy [Halpert et al. 2007], which soundly overapproximates the set of shared variables in a program (i.e., some nonshared variables might be marked shared). At instrumentation time, Symbiosis logs the code point of each shared variable access. During the symbolic execution, whenever JPF attempts to read or write a variable, it consults the log to check whether that variable is shared. If so, then JPF treats the variable as symbolic.

Comparing the two approaches, manually identifying the shared variables is clearly more complex and tedious than employing a static analysis, as it requires a careful inspection of the code. Nevertheless, we opted for following the former approach in our prototype for C/C++ applications, because we were not familiar with a thread escape analysis, similar to that of Soot, for this kind of program. Alternatively, one could employ static data race detectors to identify shared variables [Voung et al. 2007], although these often suffer from false positives.

Locks Held at Failure Points. If a thread holds a lock when it fails, then a reordering of operations in the critical region protected by the lock may lead to a deadlocking schedule. Other threads will wait indefinitely attempting to acquire the failing thread's held lock because the failing thread's execution trace includes no release. We skirt this

problem by adding a *synthetic* lock release for each lock held by the failing thread at the failure point. The synthetic releases allow the failing thread's code to be reordered without deadlocks.

4.3. Schedule Generation and DSPs

We implemented failing and alternate schedule generation, as well as differential schedule projections, from scratch in around 4K lines of C++ code. After building the SMT constraint formula, Symbiosis solves it using Z3 [De Moura and Bjørner 2008]. Symbiosis then parses Z3's output to obtain the solution of the model, or the UNSAT core, when generating the root-cause subschedule. Finally, to pretty-print its output, Symbiosis generates a graphical report (using Graphviz[7]) showing the differences between the failing and the alternate schedules.

5. EVALUATION

Our evaluation of Symbiosis focuses on answering the following three questions:

(1) How efficient is Symbiosis in collecting path profiles and symbolic path traces? (Section 5.1)
(2) How efficient is Symbiosis in solving its SMT constraint formulae? (Section 5.2)
(3) How useful are DSPs to diagnose and fix concurrency bugs? (Section 5.3)

We substantiate our results with characterization data and several case studies, using buggy, multithreaded C/C++ and Java applications, including both real-world and benchmark programs. We used four C/C++ test cases: *crasher*, a toy program with an atomicity violation; *stringbuffer*, a C++ implementation of a bug in the Java JDK1.4 `StringBuffer` library, developed in prior work [Flanagan and Qadeer 2003]; *bbuf*, a shared buffer implementation [Huang et al. 2013]; *pfscan*, a real-world parallel file scanner adapted for research by Elmas et al. [2013]; and *pbzip2*, a real-world, parallel bzip2 compressor.[8] We used four Java programs: *cache4j*, a real-world Java object cache, driven externally by concurrent update requests, and three tests from the IBM ConTest benchmarks [Farchi et al. 2003]: *airline*, *bank*, and *2stage*. Columns 1–4 of Table I describe the test cases.

We evaluated the scalability of Symbiosis for *pbzip2* and *cache4j* by varying the size of their workload. For *pbzip2*, we compressed input files of different sizes: 80KB (small), 2.6MB (medium), and 16MB (large). For *cache4j*, we reran its test driver for update loop iteration counts of 1 (small), 5 (medium), and 10 (large). In some cases, we inserted calls to the `sleep` function, changing event timing and increasing the failure rate. Our work is not targeting the orthogonal failure reproduction problem [Huang et al. 2013], so this change does not taint our results. We ran our C/C++ experiments on an 8-core, 3.5GHz machine with 32GB of memory, running Ubuntu 10.04.4. For Java we used a dual-core i5, 2.8GHz CPU with 8GB of memory, running OS X.

5.1. Trace Collection Efficiency

We measured the time and storage overhead of path profiling relative to native execution and the time cost of symbolic trace collection. Columns 6–10 of Table I report the results, averaged over five trials. Symbiosis imposes a tolerable path profiling overhead, ranging from 1.3% in *pbzip2 (medium)* to 25.4% in *crasher*. Curiously, the runtime slowdown is smaller for real-world applications (*pfscan, pbzip2, and cache4j*) than for benchmarks. The reason is that the latter programs have more basic blocks with very few operations, making block instrumentation frequent. The space overhead

[7]http://www.graphviz.org.
[8]In our experiments, we used a C version of pbzip2 from previous work Kasikci et al. [2012].

Table I. Benchmarks and Performance
Column 2 shows lines of code. Column 3 shows the number of threads. Column 4 shows the number of shared program variables. Column 5 shows the number of accesses to shared variables. Column 6 shows the overhead of path profiling. Column 7 shows the size of the profile in bytes. Column 8 shows the symbolic execution time. Column 9 shows the number of SMT constraints. Column 10 shows the number of unknown SMT variables. Column 11 shows the time in seconds to solve the SMT system.

	Application	LOC	#Threads	#Shared Vars.	#Shared Accesses	Profiling Overhead	Log Size	Symbolic Time	#SMT Conts.	#SMT Vars.	SMT Time
C/C++	crasher	70	6	4	266	25.4%	458B	0.02s	22,295	400	1m2s
	sbuff	151	2	5	69	16.7%	632B	0.05s	423	102	1s
	bbuff	377	5	11	143	34.4%	920B	13s	2,710	239	8s
	pfscan	830	5	9	74	6.6%	3.8K	1.87s	678	131	1s
	pbz2 (S)	1,942	9	14	176	2.5%	1.7K	11.16s	1,361	289	1s
	pbz2 (M)				367	1.3%	2.6K	36.17s	6,771	564	26s
	pbz2 (L)				1,156	2.5%	9.4K	7m11s	514,548	2,866	15h15m
Java	airline	108	8	2	36	22%	262B	1.30s	2,670	84	1s
	bank	125	3	3	115	12.4%	788B	1.56s	8,250	197	2s
	2stage	123	4	4	49	14.8%	196B	2.53s	264	88	1s
	c4j (S)	2,344	4	7	28	7.3%	366B	1.64s	122	51	1s
	c4j (M)				1,247	8.6%	17K	4.56s	303,626	1,810	51s
	c4j (L)				1,411	9.3%	24K	4.76s	1,142,120	2,051	1h 25m

of path profiling is also low, ranging from 196B (*2stage*) to 24K (*cache4j*). CLAP [Huang et al. 2013] showed that recording threads' path profiles only reduces storage overheads considerably (up to 97%!) compared to R&R (e.g., LEAP [Huang et al. 2010]). Symbiosis enjoys this reduction as well. Symbiosis collects symbolic traces in just seconds for most test cases. The only exception is *pbzip2 (large)*, which took KLEE around 7 minutes. JPF quickly produced the symbolic traces for all programs.

5.2. Constraint System Efficiency

The last three columns of Table I describe the SMT formulae Symbiosis built for each test case. The table also reports the amount of time Symbiosis takes to solve its SMT constraints with Z3, yielding a failing schedule. The data show that solver time is very low (i.e., seconds) in most cases. Solver time often grows with constraint count but not always. *cache4j (large)* has more than double the constraints of *pbzip2 (large)* but was around 11 times faster. Figure 5 helps explain the discrepancy by showing the composition of the SMT formulations by constraint type. *pbzip2* has many *locking* and *read-write* constraints, while *cache4j* has no *locking* although many *read-write* constraints. The solution to locking constraints determines the execution's lock order, constraining the solution to read-write constraints. The formulation's complexity grows not with the count but the interaction of these constraint types.

Symbiosis's SMT solving times are practical for debugging use. To produce a DSP, Symbiosis requires only a trace from a single, failing execution and does not require any changes to the code or input. Our experiments are realistic because a programmer, when debugging, often has a bug report with a small test case that yields a short, simple execution. The data suggest that Symbiosis handles such executions very quickly (e.g., *pbzip2 (small)*, *cache4j (medium)*). Debugging is a relatively rare development task, unlike compilation, which happens frequently. Giving Symbiosis minutes or hours to help solve hard bugs (like *pbzip2 (large)*) is reasonable. Additionally, Symbiosis could divide the SMT constraint system into different instances and solve them in parallel, like CLAP, or incorporate lock ordering information, like [Bravo et al. 2013], to decrease solver time.

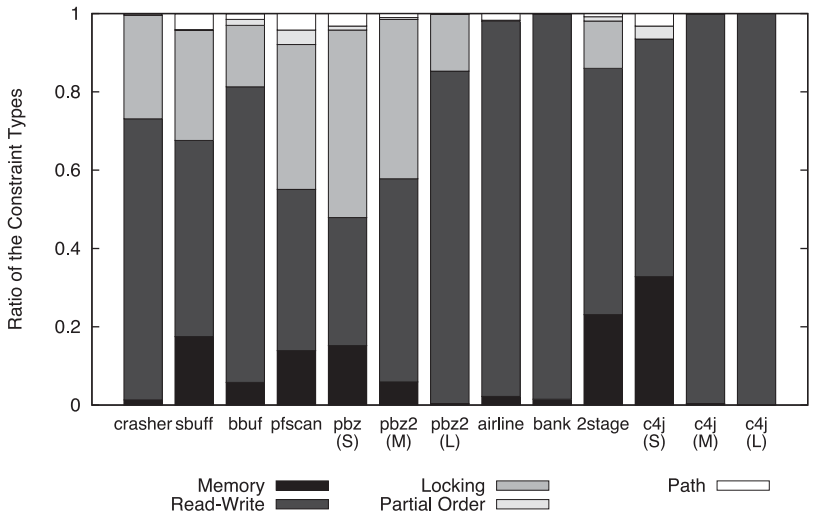

Fig. 5. Breakdown of the SMT constraint types.

5.3. Differential Schedule Projections Effectiveness

Symbiosis produces a graphical visualization of its DSPs as a graph with specific identifying information on nodes and edges that reflects source code lines and variables. This information includes schedule variations and dataflow variations as well.

In this section, we are interested in assessing the effectiveness of DSPs to diagnose concurrency failures. To this end, we first evaluate the efficacy of DSPs in isolating the root cause of the benchmark bugs. Second, we use case studies to further illustrate how DSPs can be used to understand and fix those bugs. Third, we investigate the impact of using context switch reduction when generating DSPs. Finally, we present a user study that assesses the benefits of DSPs over full failing schedules for concurrency bug diagnosis.

5.3.1. Root-Cause Isolation Efficacy. To evaluate the efficacy of DSPs in isolating the bug's root cause, we compared the number of program events and dataflow edges in the differential schedule projection against those of full, failing executions computed by Symbiosis.

Table II summarizes our results. The most important result is that Symbiosis's differential schedule projections are simpler and clearer than looking at full, failing schedules. Symbiosis reports a small fraction of the full schedule's dataflows and program events in its output—on average, 90% fewer events and 96% fewer dataflows. By highlighting only the operations involved in the dataflow variations, Symbiosis focuses the programmer on just a few events (three to six in our tests). Furthermore, all events Symbiosis reports are part of dataflow or event orders that dictate the presence or absence of the failure. DSPs depict those events only, simplifying debugging.

Symbiosis finds an alternate, nonfailing schedule after reordering few event pairs—just 1 in many cases (e.g., *cache4j*, *pbzip2*). Symbiosis reorders one pair at a time, starting from those closer in the schedule to failure, and the data show that this usually works well. *bank* is an outlier—Symbiosis reordered 181 different pairs before finding an alternate, nonfailing schedule. The bug in this case is an atomicity violation that breaks a program invariant that is not checked until later in the execution. As a result, Symbiosis must search many pairs, starting from the failure point, to eventually reorder the operations that cause the atomicity violation.

Table II. Differential Schedule Projections
Columns 2 is the number of event pairs reordered to find a satisfiable alternate schedule (*#Alt Pairs*). Column 3 shows the number of events in the failing schedule (*#Events in Fail Sch.*), and Column 4 shows the number of events in the corresponding differential schedule projection (*#Events DSP*). Column 5 shows the number of dataflow edges in the failing schedule (*#D-F in Fail Sch.*), and Column 6 shows the number of dataflow variations in the differential schedule projection (*#D-F in DSP*). Columns 4 and 6 show the percent change compared to the full schedule. Column 7 shows the number of operations involved in the dataflow variations (*#Ops to Grok*). Columns 8 and 9 show whether the differential schedule projection explains the failure, and whether it directly points to a fix of the underlying bug in the code.

Application	#Alt. Pairs	#Events in Fail Sch.	#Events in DSP (Δ%)	#D-F in Fail Sch.	#D-F in DSP (Δ%)	Ops. to Grok	Explanatory?	Finds Fix?
crasher	27	287	9 (↓**97**)	107	1 (↓**99**)	3	Y	Y
sbuff	9	73	15 (↓**80**)	28	1 (↓**96**)	3	Y	Y
bbuff	3	157	19 (↓**88**)	79	1 (↓**99**)	3	Y	Y
pfscan	5	93	16 (↓**83**)	32	1 (↓**97**)	3	Y	Y
pbz2 (S)	1	206	4 (↓**98**)	29	1 (↓**97**)	3		
pbz2 (M)	1	397	3 (↓**99**)	82	1 (↓**99**)	3	Y	N
pbz2 (L)	2	1223	168 (↓**86**)	264	2 (↓>**99**)	5		
airline	1	58	6 (↓**90**)	25	2 (↓**92**)	6	Y	Y
bank	181	124	31 (↓**75**)	72	2 (↓**97**)	5	Y	Y
2stage	14	60	3 (↓**95**)	27	1 (↓**96**)	3	Y	Y
c4j (S)	1	39	12 (↓**69**)	11	2 (↓**82**)	6		
c4j (M)	1	1257	10 (↓>**99**)	552	1 (↓>**99**)	3	Y	N
c4j (L)	1	1422	5 (↓>**99**)	628	1 (↓>**99**)	3		

Note that even if a failure occurs only in the presence of a particular chain of event orderings, it suffices to reorder any pair in the chain to prevent that failure. This phenomenon is called the *Avoidance-Testing Duality* and is detailed in previous work [Lucia and Ceze 2013].

5.3.2. Differential Schedule Projections Case Studies. This section uses case studies to illustrate how differential schedule projections focus on relevant operations and help understand each failure.

stringbuffer is an atomicity violation first studied in Flanagan and Qadeer [2003] and its DSP is depicted in Figure 6(a). *T1* reads the length of the string buffer, *sb*, while *T2* modifies it. When *T2* erases characters, the value *T1* read becomes stale and *T1*'s assertion fails. The DSP shows that the cause of the failure is *T2*'s second write, interleaving *T1*'s accesses to *sb.count*. Moreover, Symbiosis's alternate schedule suggests that, for *T1*, the write on value *len* and the verification of the assertion should execute atomically in order to avoid the failure. For this case, this is actually a valid bug fix.

bbuf contains producer/consumer threads that put/get items into/from a shared buffer for a given number of times. This program has an atomicity violation that allows consumer threads to get items from the buffer, even when it is empty. Figure 6(b) illustrates this failure: after getting an item from the buffer, consumer thread *T1* prepares to get another one, but first checks whether the buffer is empty (i.e., `if(bbuf->head!=bbuf->rear)`). As the condition is true, *T1* proceeds to consume the item, but it is interleaved by *T2* in the meantime, which consumes the item first and updates the value of `bbuf->head`. This causes *T1* to later violate the assertion that enforces the buffer invariant. The alternate schedule in Figure 6(b) shows that executing atomically the two blocks that, respectively, check the conditional clause and the assertion prevents the failure.

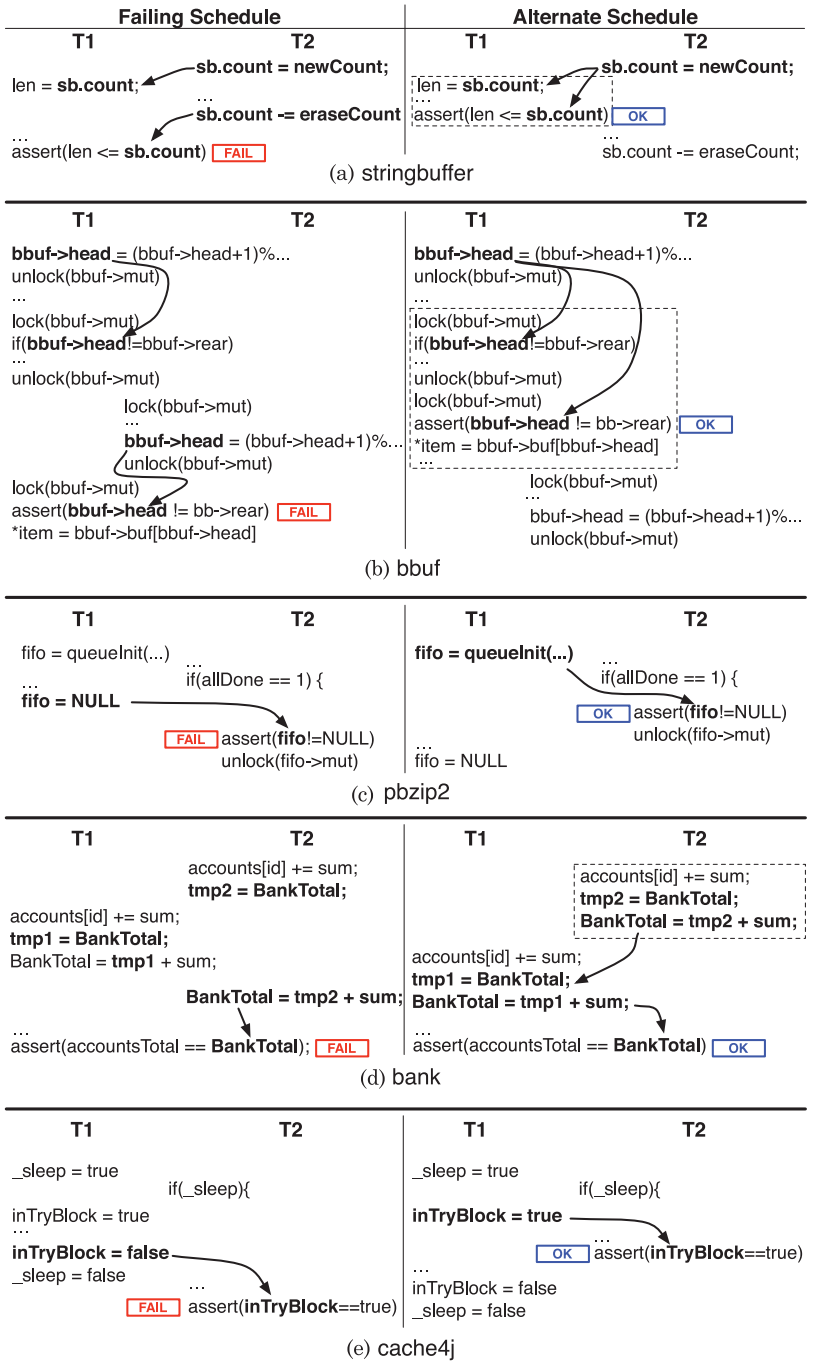

Fig. 6. Summary of Symbiosis's output for some of the test cases. Arrows depict dataflows and dashed boxes depict regions that Symbiosis suggests to be executed atomically.

pbzip2 is an order violation studied in Jalbert and Sen [2010]. Figure 6(c) shows Symbiosis's DSP that illustrates the failure's cause. *T1*, the producer thread, communicates with *T2* the consumer thread via the shared queue, *fifo*. If *T1* sets the *fifo* pointer to null while the consumer thread is still using it, then *T2*'s assertion fails. The alternate schedule in Figure 6(b) explains the failure because reordering the assignment of *null* to *fifo* after the assertion prevents the failure. The DSP is, thus, useful for understanding the failure. However, to fix the code, the programmer must order the assertion with the null assignment using a `join` statement. The DSP does not provide this suggestion, so, despite helping explain the *failure*, it does not completely reveal how to fix the bug.

bank is a benchmark in which multiple threads update a shared bank balance. It has an atomicity violation that leads to a lost update. Figure 6(d) shows the DSP for the failure: *T1* and *T2* read the same initial value of *BankTotal* and subsequently write the same updated value, rather than either seeing the result of the other's update. The final assertion fails, because *accountsTotal*, the sum of per-account balances, is not equal to *BankTotal*. The Figure shows that Symbiosis's DSP correctly explains the failure and shows that eliminating the interleaving of updates to *BankTotal* prevents the failure. It is noteworthy that in this example the atomicity violation is not *fail-stop* and happens in the middle of the trace. Scanning the trace to uncover the root cause would be difficult, but the DSP pinpoints the failure's cause precisely.

cache4j has a data race that leads to an uncaught exception when one thread is in a `try` block and another interrupts it with the library `interrupt()` function [Sen 2008]. JPF doesn't support exception replay, so we slightly modified its code, preserving the original behavior, by replacing the exception with an assertion about a shared variable. Figure 6(e) shows that in our version of the code, *inTryBlock* indicates whether a thread is inside a `try-catch` block, the assertion `inTryBlock == true` replaces the `interrupt()` call. The program fails when *T1* is interrupted outside a `try` block as in the original code. The schedule variations reported in the DSP explain the cause of failure—if the entry to the `try` block (i.e., `inTryBlock = true`) precedes the assertion, execution succeeds; if not, the assertion fails. The involvement of exceptions makes the fix for this bug somewhat more complicated than simply adding atomicity, but the understanding that the DSP provides points to the right part of the code and illustrates the correct behavior.

5.3.3. Impact of Context Switch Reduction. We evaluate the impact of the CSR algorithm presented in Section 3.7. To this end, we ran Symbiosis with and without CSR and compared: (i) the number of context switches of the failing schedule generated, (ii) the number of event pairs reordered to find a satisfiable alternate schedule, and (iii) the number events and dataflows in the DSPs produced.

Table III reports the results of our experiments. The most prominent observation is that our CSR algorithm is indeed effective in reducing the number of context switches (the failing schedules have 63% less context switches, on average). Table III also shows that, when using CSR, Symbiosis is able to find a satisfiable alternate schedule with less event pair reorderings in three of the test cases (*crasher*, *pfscan*, and *bank*).

On the other hand, DSPs produced by Symbiosis with CSR tend to have slightly more events than produced without CSR. The reason is because CSR produces schedules with more coarse-grained thread segments (i.e., comprising more events) and our current DSP implementation does not eliminate events from the same thread segment that occur in between two events involved in dataflow variations.

Another observation worth noting from Table III is that DSPs with CSR do not exhibit any reductions in terms of dataflow variations with respect to DSPs without CSR. The reason is because the feasible alternate schedules produced by Symbiosis are

Table III. Context Switch Reduction Efficacy
Column *#CS* indicates the number of schedule context switches; *#Alt Pairs* is the number of event pairs reordered to find a satisfiable alternate schedule; *#Events in DSP* and *#D-F in DSP* show, respectively, the number of events and number of dataflow in the corresponding differential schedule projection. *Time CSR* indicates the total amount of time required to perform the context switch reduction algorithm. Shaded cells indicate the cases where Symbiosis achieves better results with CSR than without.

	Without CSR				With CSR				
Application	#CS	#Alt. Pairs	#Events in DSP	#D-F in DSP	#CS	#Alt. Pairs	#Events in DSP	#D-F in DSP	Time CSR *(Solver Calls)*
crasher	104	27	9	1	34	21	9	1	19m 33s *(166)*
sbuff	7	9	15	1	4	9	15	1	2s *(14)*
bbuff	35	3	19	1	10	6	19	1	14s *(40)*
pfscan	23	5	16	1	9	3	16	1	9s *(32)*
pbz2 (S)	74	1	4	1	14	1	7	1	14s *(59)*
pbz2 (M)	151	1	4	1	22	1	7	1	4m 58s *(163)*
pbz2 (L)	292	1	4	1	36	1	7	1	5h 11m *(353)*
airline	28	1	6	2	8	1	8	2	5s *(33)*
bank	6	181	31	2	4	154	46	2	6s *(9)*
2stage	16	14	3	1	4	14	6	1	2s *(15)*
c4j (S)	4	1	12	2	4	1	12	2	<1s *(2)*
c4j (M)	21	1	10	1	7	1	10	1	6m 50s *(25)*
c4j (L)	29	1	5	1	7	1	5	1	16m 19s *(29)*

mainly the result of reordering a event from a thread (typically the one corresponding to the failure condition) with an event from another thread that is close in the schedule. Hence, most dataflows do not change after reordering the event pair, even if the failing schedule has unnecessary context switches.

Regarding the amount of time required to perform CSR (shown in the last column of Table III), it is possible to see that Symbiosis took only a couple of seconds for most cases. However, for *pbzip2 (L)* this time exceed 5 hours. The reason is because the failing schedule for this program contained a significant number of context switches, which required the CSR to invoke the solver several times (353 to be precise). Moreover, since the constraint model for *pbzip2 (L)* is also the one with most constraints, each solver call becomes particularly costly.

In conclusion, despite CSR being effective in reducing the number of context switches in a failing schedule, our experiments show that this does not imply a reduction in the number of events and dataflow variations reported in the DSPs. Furthermore, since performing CSR can be costly for some programs, we argue that computing the DSP with the original failing schedule should probably be the most cost-effective approach for the majority of the cases.

5.3.4. User Study. To assess how useful for debugging a DSP is, compared to full failing schedule, we conducted a user study.

Participants. We recruited 48 participants, including 21 students (6 undergraduate, 9 masters, 6 doctoral) from Instituto Superior Técnico, 24 masters students from University of Pennsylvania, 1 doctoral student from Carnegie Mellon University, and 2 software engineers (with 3 years of experience in the industry), to individually find the root cause of a given concurrency bug.

Study Design. The participants were randomly divided into two groups according to the type of debugging aid they were going to use in the experiment: the full schedule of a failing execution or the DSP for the same failing schedule.

Fifteen participants took the study in a proctored session. The remaining 33 participants received the study by email and returned it by email on completion.

Prior to initiating the experiment, each participant had access to a tutorial example that explained how the respective debugging aid could be used to find the root cause of a concurrency bug in a toy program. The goal of the tutorial was to guarantee that each participant knew how to read and understand its debugging aid (i.e., the full failing schedule or the DSP) beforehand.

For the experiment, we provided each participant with the source code of a multithreaded program with a concurrency bug. This bug was a simplified version of the *stringbuffer* error used in the previous sections and consisted of an atomicity violation that caused the failure of an assertion (the assertion included two conditions of which only one fails). In addition, each participant was given its corresponding debugging aid: the full schedule of a failing execution or the DSP for the same failing schedule.

The experiment consisted of analyzing the debugging aid and the program's source code and answering a short survey with five questions. First, we asked which conjunct of the assertion condition was violated (allowing us to screen for wrong answers). Second, we asked the participant to write up to three sentences describing what caused the assertion to fail. Then, we asked the participant to rate, on a scale of 1 to 5, the difficulty of diagnosing the concurrency bug, as well as their experience in debugging multithreaded programs. Finally, we asked the participant to report the time they took to find the bug by choosing one of four intervals of time: *0–4min*, *5–9min*, *10–14min*, and $\geq$*15min*.

Results. We analyze these results according to three different criteria, which are discussed below.

—*Correctness.* Do the participants correctly identified the root cause of the assertion violation? Does the type of debugging aid have an impact in the success rate?
—*Bug Difficulty.* Does the type of debugging aid have an impact in the self-reported bug difficulty?
—*Diagnosis Time.* Does the type of debugging aid have an impact in the time required to diagnose the bug? Are there other factors that significantly influence the diagnosis time (e.g., the participant's debugging experience)?

To support the conclusions of the user study, we performed a statistical analysis over the obtained data. Concretely, for each criterion, we started by computing the correlation coefficient between the variables being analyzed (e.g., identifying the correct root cause of the bug and using DSP as debugging aid). For the cases where the correlation coefficient indicated a statistically significant relationship between the variables, we also performed a *t-test* over the samples to further support that claim.

Correctness. We considered that participants had a correct answer when they correctly identified the failing conjunct in the assertion condition and provided a satisfactory explanation to the bug's root cause.

From the 48 participants, 38 answered correctly. In particular, the participants that received the DSP show a slightly higher percentage of correct answers in comparison to those who received the full failing schedule (74% against 71%, respectively).

To statistically evaluate the relationship between the type of debugging aid and the correctness of the answer, we calculated the *point-biserial correlation coefficient*[9]

[9]We opted for using the point-biserial correlation coefficient to compute the correlation because the debugging aid variable is *naturally dichotomous*, that is, it corresponds to either using the DSP or the full failing schedule. The variable representing the correctness of the answer is also naturally dichotomous, as the answer can only be considered *correct* or *incorrect*. Note that the correlation coefficient varies between -1

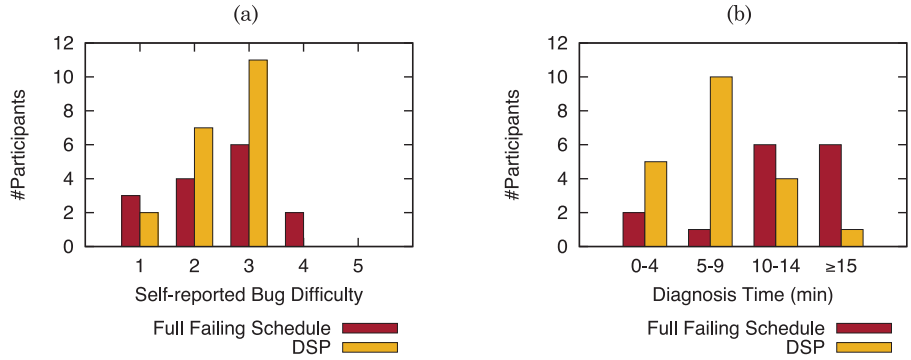

Fig. 7. User Study Results (considering solely correct answers). (a) Impact of the type of debugging aid in self-reported bug difficulty (1 means very easy, 5 means very hard); (b) impact of the type of debugging aid (full schedule/DSP) in diagnosis time.

between these two variables. We considered the variable debugging aid to have value 0 for the DSP and 1 for the full failing schedule. For the correctness of the answer, we considered 0 to indicate that the response is incorrect and 1 to indicate that it is correct.

The correlation value between the debugging aid variable and the correctness of the answer is −0.030, which means that there is no statistically significant correlation between the two variables. The negative value of the coefficient shows that there is a slight trend for people using the DSP to answer correctly more often than people with full failing schedules though.

Given that the majority of the participants successfully found the root cause of the bug, we believe that, for (somewhat) simple concurrency bugs, having any kind of debugging aid is indeed helpful for debugging.

Bug Difficulty. We are interested in understanding whether participants doing the experiment with DSPs would find the bug easier to debug. Figure 7(a) depicts the values for bug difficulty reported by the participants according to their type of debugging aid. From the figure, it is not possible to extract a clear trend, although we can see that no participant using DSPs rated the bug above 3, whereas two participants with the full schedule classified the bug as 4.

We computed the correlation coefficient for these two variables and obtained the value 0.010, which indicates there is indeed no statistically significant correlation between the type of debugging aid and the bug difficulty. However, the value of the coefficient points out a slim positive relationship between using the full schedule and finding the bug harder to diagnose.

Similarly, the correlation coefficient for the participants' experience and the bug difficulty (which has value −0.048) shows that there is no statistically significant correlation between these two variables, although the tendency for this case is that participants with more experience tend to find the bug easier than less experienced participants.

Diagnosis Time. Figure 7(b) reports the participants' diagnosis time according to the type of debugging aid (full schedule or DSP) they received. The figure shows that participants that received the DSP tended to diagnose the bug in less time. To statistically

and 1, where −1 (1) indicates a very strong negative (positive) correlation between the variables, and 0 indicates that there is no correlation at all between the variables.

evaluate this claim, we calculated the point-biserial correlation coefficient between type of debugging aid and the amount of time to diagnose the bug. Once more, we considered the variable debugging aid to have value 0 for the DSP and 1 for the full failing schedule.

The correlation value between the debugging aid variable and diagnosis time is 0.416, which means that, with a 99% confidence level, there is a significant positive relationship between these two variables. This means that using the full failing schedule is correlated to a greater amount of time to find the bug. In other words, the correlation supports the claim that using a DSP allows for faster bug diagnosis, as we expected.

We also performed a *Welch's t-test* to test whether the means of our two samples (the DSP sample vs. the full schedule sample) differ statistically significantly. A significant result means that the sampled means correspond to different underlying populations, and, as the data show, that the DSP sample mean is less than the failing schedule sample mean. Let M_0 be the mean of the diagnosis times for the DSP tests and M_1 the mean of the diagnosis times for the full failing schedule tests. We consider the null hypothesis to be "$M_0 = M_1$," and we test whether to reject the hypothesis.

The t-test's two-tailed value of t_p with 24 degrees of freedom[10] and a 95% confidence level ($p < 0.05$) is 2.064. Evaluating the Welsh's t-test, $t = -2.281$. Since $2.064 < |-2.281|$ and, consequently, $t_p < t$, we can reject the null hypothesis and conclude that $M_0 \neq M_1$. This result shows that there is a statistically significant decrease in debugging time using a DSP, compared to using a failing schedule.

Finally, we computed the Pearson's correlation coefficient between participants' self-reported experience in debugging multithreaded programs and the diagnosis time, in order to assess whether the experience was also a significant factor in the diagnosis time. The correlation coefficient for this case showed that there was only an extremely weak, negative relationship between debugging experience and diagnosis time. This means that participants who reported higher experience tend to find the bug slightly faster than participants with less experience, although the difference is not statistically significant.

In summary, regarding the benefits of DSPs over full failing schedules, the main findings of our user study show that:

—*Correctness.* There is a slim correlation between using DSPs and a correct bug diagnosis, although it is not statistically relevant.
—*Bug Difficulty.* There is a slight correlation between using DSPs and finding the bug easier to debug, although it is not statistically relevant.
—*Diagnosis Time.* There is a *statistically significant* correlation between using DSPs and a faster diagnosis time.

Thus, in our study, all users had a similar perception of the difficulty of the bug at hand (which is not surprising, given that the intrinsic "hardness" of a bug is somehow independent of the tools used to find it). Furthermore, in both groups, approximately the same percentage of users were able to found the correct answer (which suggests that both groups had a similar ability/experience to recognize the right answer). Still, the group using DSPs was able to perform the diagnosis faster, which supports our claim that Symbiosis can reduce the debugging time.

Threats to validity. To simplify the study, we designed the whole experiment to be supported solely by textual material. As such, we crafted the experiment's concurrency bug in such a way that it could be solved in a practical 20 minutes by a person not familiar with the program's source code. This fact might have diminished the debugging

[10]The number of degrees of freedom was calculated using the *Welch-Satterthwaite equation* for the Welch's t-test.

time difference between using a DSP and a full schedule because all participants, regardless of debugging aid, may have taken substantial time to understand the code. We expect that the observed difference in debugging time would be greater if the participant was already familiar with the code, eliminating the fixed time cost for understanding the code.

The diagnosis time for the participants in the offsite (email) setting was self-reported, which might be subject to some inaccuracies. Despite that, we observed that the results from the offsite setting are consistent with those onsite (the proctored session).

Since the self-reported debugging experience was not measured using exact metrics, it might be biased towards each participant's self-perception of what it means to be an expert in debugging multithreaded programs. In fact, the values obtained do not always match with education level of the participants (for instance, there was a doctoral student who reported an experience level of 1 and an undergraduate who reported an experience level of 3). To mitigate this threat, we computed the correlation coefficients for the diagnosis time and the bug difficulty using the education level instead of the self-reported experience. We observed similar results and, thus, concluded that self-reported experience was a valid factor to take into account.

6. RELATED WORK

In addition to the work discussed in Section 2, a large body of prior research has been devoted to debugging of multithreaded programs. This section overviews some of the previous efforts most related to Symbiosis.

Record and Replay. *Record and Replay* (R&R) techniques are also relevant to our work. These techniques fall into three categories: *order-based*, *search-based*, and *execution-based*.

Order-based techniques record the order of certain events during an execution and then replay them in the same order [Huang et al. 2010; Yang et al. 2011; Zhou et al. 2012; Jiang et al. 2014]. Search-based techniques only trace partial information at runtime (e.g., record solely the order of write operations [Zhou et al. 2012]) and then search the space of executions for one that conforms with the observed events [Zamfir and Candea 2010a; Park et al. 2009; Altekar and Stoica 2009; Machado et al. 2012]. Execution-based techniques restrict all executions of a program so, for a given input, the program's behavior is constrained to be deterministic from one run to the next [Berger et al. 2009; Liu et al. 2011; Devietti et al. 2009; Bergan et al. 2010].

Symbiosis is mostly orthogonal to the techniques above but shares some important characteristics. Like R&R techniques, given a concrete trace, Symbiosis can produce a failing schedule that conforms to those events, reproducing the failure. Symbiosis's precise differential schedule projections and broader applicability to debugging and failure avoidance make it novel in contrast to R&R techniques. Unlike deterministic execution techniques, Symbiosis does not aim to perturb production runs, obviating the risk in doing so.

Concurrency Debugging and Failure Avoidance. Several techniques have been proposed over the past few years to identify the root cause of a concurrency bug, show diagnose information to help the programmer fix it, or to avoid it in future executions. We discuss some of the most relevant research efforts on this matter in the following.

In the particular case of data race detection, Portend [Kasikci et al. 2012] and the work by Narayanasamy et al. [2007] provide the developer with information regarding which data races reported by a data race detector are *harmful* or *benign*. This way, developers can focus their efforts on fixing the racy accesses that can actually lead to failures. Symbiosis can also help diagnose and fix harmful data races in the presence of failing schedules containing this kind of concurrency bug.

Interleaving pattern-matching [Park et al. 2010; Lucia et al. 2010; Lu et al. 2006] techniques search an execution, dynamically or by reviewing a log, for problematic patterns of memory accesses. Although often effective, these solutions have the drawback of missing bugs that not fit the known patterns. Unlike these techniques, Symbiosis is not limited to searching for known patterns.

Subschedule search solutions, in turn, are general and not limited to specific patterns [Zhang et al. 2011; Lucia and Ceze 2013; Shi et al. 2010]. Unfortunately, the space of all of an execution's possible subschedules can be large. For some prior techniques, considering different subschedules requires multiple additional program executions, combined with statistical analysis to make search feasible [Arumuga Nainar and Liblit 2010; Jin et al. 2010; Kasikci et al. 2015]. Symbiosis requires only a single, failing execution; does not rely on statistical reasoning; and produces precise results. Mechanically, these techniques differ in that none uses SMT to search and none produces a differential view of its result, like DSPs.

PBI [Arulraj et al. 2013] and LBRA/LCRA [Arulraj et al. 2014] rely on custom hardware extensions, such as performance counters and short-term memory, to diagnose production-run failures caused by sequential and concurrency bugs with low overhead. However, these techniques also require sampling several production runs to be effective and only work well in the presence of bugs where the root cause is close to the failure.

Triage [Tucek et al. 2007] uses dynamic slicing to diagnose failures at the user's site, which obviates privacy concerns. Despite that, it has limited support for concurrency bugs, being able to provide root cause isolation only for multithreaded programs running on uniprocessors.

Root-Cause Isolation with UNSAT Cores. BugAssist [Jose and Majumdar 2011] pioneered the use of UNSAT cores to isolate errors in software programs. BugAssist supports only sequential programs and requires several failing test cases, whereas Symbiosis is aimed at diagnosing concurrency bugs, even in the presence of a single failing schedule.

Contemporaneously to Symbiosis, ConcBugAssist [Khoshnood et al. 2015] extended BugAssist to handle bugs in multithreaded programs. ConcBugAssist also relies on the UNSAT core feature to compute the subset of constraints that comprise the root cause of a concurrency bug. However, instead of producing DSPs, ConcBugAssist attempts to generate automated repairs by casting the *binate covering problem* as a constraint formulation. This process has the drawback of requiring model checking the entire program and compute all possible schedules that prevent the failure, which is hard to do in practice.

Testing Approaches. Finally, other techniques systematically explore the space of possible program executions to generate test cases. Java Path Finder [Visser et al. 2004], KLEE [Cadar et al. 2008], Pex [Tillmann and De Halleux 2008], and Mimic [Zuddas et al. 2014] use symbolic program execution to search for an *input* that induces a failing path constraint. Chess [Musuvathi et al. 2008], PCT [Burckhardt et al. 2010], and Concurrit [Elmas et al. 2013] run a program for a particular input and rely on an augmented scheduler to push the execution to a potential failure. On the other hand, con2colic testing [Farzan et al. 2013], CUTE [Sen et al. 2005], and DART [Godefroid et al. 2005] employ *concolic execution* (i.e., *conc*rete + symb*olic* execution), which is a technique that uses concrete input values to simplify complex symbolic constraints. In particular, CUTE and DART focus on the generation of unit tests for sequential programs, whereas con2colic testing uses heuristics to explore the space of possible thread interleavings and execution paths to generate tests for multithreaded programs.

These techniques reveal only full, failing executions or buggy inputs and provide neither root-cause information nor differential schedule projections. In turn, Choi and

Zeller [2002] also shares our goal of narrowing down the difference between successful and failing schedules to pinpoint a bug. This technique relies on random jitter and requires reexecuting the program, whereas Symbiosis only operates on SMT formulations, which are sound and complete and, thus, provide a formal guarantee that alternate schedules are nonfailing.

7. CONCLUSIONS AND FUTURE WORK

This article described Symbiosis, a new technique that gets to the bottom of concurrency bugs. Symbiosis reports focused *subschedules*, eliminating the need for a programmer or automated debugging tool to search through an entire execution for the bug's root cause. Symbiosis also reports novel *alternate, nonfailing schedules*, which help illustrate *why* the root cause is the root cause and how to avoid failures. Our novel *differential schedule projection* approach links the root cause and alternate subschedules to dataflow information, giving the programmer deeper insight into the bug's cause than path information alone. An essential part of Symbiosis's mechanism is the use of an SMT solver and, in particular, its ability to report the part of a formula that makes it unsatisfiable. Symbiosis carefully constructs a deliberately unsatisfiable formula so the conflicting part of that formula is the bug's root cause. We built two Symbiosis prototypes, one for C/C++ and one for Java. We used them to show that, for a variety of real-world and benchmark programs from the debugging literature, Symbiosis isolates bugs' root causes and providing differential schedule projections that show how to fix those root causes.

As future research directions, we plan to extend Symbiosis to support the diagnosis of concurrency bugs that result from schedule-sensitive branches [Huang and Rauchwerger 2015]. Additionally, we also intend to extend Symbiosis's constraint-based approach to expose new concurrency bugs from correct production run traces rather than just isolating the root cause of a failing schedule.

ACKNOWLEDGMENTS

We thank both PLDI'15 and TOSEM anonymous reviewers for their valuable and constructive feedback. We also thank Baris Kasikci for kindly sharing the C version of pbzip2 with us. Finally, a special thank you to all the participants for their help with the user study.

REFERENCES

Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 193–206. DOI: http://dx.doi.org/10.1145/1629575.1629594

Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 101–112. DOI: http://dx.doi.org/10.1145/2451116.2451128

Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 207–222. DOI: http://dx.doi.org/10.1145/2541940.2541973

Piramanayagam Arumuga Nainar and Ben Liblit. 2010. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE'10)*. ACM, New York, NY, 255–264. DOI: http://dx.doi.org/10.1145/1806799.1806839

Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1319–1360. DOI: http://dx.doi.org/10.1145/183432.183527

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 53–64. DOI: http://dx.doi.org/10.1145/1736020.1736029

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 81–96. DOI:http://dx.doi.org/10.1145/1640089.1640096

Manuel Bravo, Nuno Machado, Paolo Romano, and Luís Rodrigues. 2013. Towards effective and efficient search-based deterministic replay. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep'13)*. ACM, New York, NY, Article 10, 6 pages.

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 167–178. DOI:http://dx.doi.org/10.1145/1736020.1736040

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

Jong-Deok Choi and Andreas Zeller. 2002. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*. ACM, New York, NY, 210–220. DOI:http://dx.doi.org/10.1145/566172.566211

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, 85–96. DOI:http://dx.doi.org/10.1145/1508244.1508255

Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 153–164. DOI:http://dx.doi.org/10.1145/2491956.2462162

Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. 2010. Encoding industrial hardware verification problems into effectively propositional logic. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD'10)*. FMCAD Inc, Austin, TX, 137–144. http://dl.acm.org/citation.cfm?id=1998496.1998522

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 237–252. DOI:http://dx.doi.org/10.1145/945445.945468

Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*. IEEE Computer Society, Washington, DC, 286–293.

Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2Colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, 37–47. DOI:http://dx.doi.org/10.1145/2491411.2491453

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 121–133. DOI:http://dx.doi.org/10.1145/1542476.1542490

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 293–303. DOI:http://dx.doi.org/10.1145/1375581.1375618

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 338–349. DOI:http://dx.doi.org/10.1145/781131.781169

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 213–223. DOI:http://dx.doi.org/10.1145/1065010.1065036

Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. 2007. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE Computer Society, Washington, DC, 353–364. DOI:http://dx.doi.org/10.1109/PACT.2007.23

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. DOI:http://dx.doi.org/10.1145/78969.78972

Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight deterministic multiprocessor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 207–216. DOI:http://dx.doi.org/10.1145/1882291.1882323

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 337–348. DOI:http://dx.doi.org/10.1145/2594291.2594315

Jeff Huang and Lawrence Rauchwerger. 2015. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, 439–449. DOI:http://dx.doi.org/10.1145/2786805.2786840

Jeff Huang and Charles Zhang. 2011. An efficient static trace simplification technique for debugging concurrent programs. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, 163–179. http://dl.acm.org/citation.cfm?id=2041552.2041567

Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 141–152. DOI:http://dx.doi.org/10.1145/2491956.2462167

Nicholas Jalbert and Koushik Sen. 2010. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 57–66. DOI:http://dx.doi.org/10.1145/1882291.1882302

Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. CARE: Cache guided deterministic replay for concurrent java programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 457–467. DOI:http://dx.doi.org/10.1145/2568225.2568236

Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, New York, NY, 241–255. DOI:http://dx.doi.org/10.1145/1869459.1869481

Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 437–446. DOI:http://dx.doi.org/10.1145/1993498.1993550

Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 344–360. DOI:http://dx.doi.org/10.1145/2815400.2815412

Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, 185–198. DOI:http://dx.doi.org/10.1145/2150976.2150997

Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. ACM, New York, NY, 165–176. DOI:http://dx.doi.org/10.1145/2771783.2771798

James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. DOI:http://dx.doi.org/10.1145/360248.360252

Shuvendu Lahiri and Shaz Qadeer. 2008. Back to the future: Revisiting precise program verification using SMT solvers. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM, New York, NY, 171–182. DOI:http://dx.doi.org/10.1145/1328438.1328461

L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. DOI:http://dx.doi.org/10.1109/TC.1979.1675439

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 327–336. DOI:http://dx.doi.org/10.1145/2043556.2043587

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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 (ASPLOS XIII). ACM, New York, NY, 329–339. DOI:http://dx.doi.org/10.1145/1346281.1346323

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, 37–48. DOI:http://dx.doi.org/10.1145/1168857.1168864

Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 553–563. DOI:http://dx.doi.org/10.1145/1669112.1669181

Brandon Lucia and Luis Ceze. 2013. Cooperative empirical failure avoidance for multithreaded programs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 39–50. DOI:http://dx.doi.org/10.1145/2451116.2451121

Brandon Lucia, Luis Ceze, and Karin Strauss. 2010. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 222–233. DOI:http://dx.doi.org/10.1145/ 1815961.1815988

Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, 277–288. DOI:http://dx.doi.org/10.1109/ISCA.2008.4

Brandon Lucia, Benjamin P. Wood, and Luis Ceze. 2011. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 378–388. DOI:http://dx.doi.org/ 10.1145/1993498.1993543

Nuno Machado, Paolo Romano, and Luis Rodrigues. 2012. Lightweight cooperative logging for fault replication in concurrent programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*. IEEE Computer Society, Washington, DC, 1–12.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 267–280.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 22–31. DOI:http://dx.doi.org/10.1145/1250734.1250738

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, Washington, DC, USA, 97–108.

Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE'10)*. ACM, New York, NY, 245–254. DOI:http://dx.doi.org/10.1145/1806799.1806838

Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 177–192. DOI:http://dx.doi.org/10.1145/1629575.1629593

Shaz Qadeer. 2009. Algorithmic verification of systems software using SMT solvers. In *Proceedings of the 16th International Symposium on Static Analysis (SAS'09)*. Springer-Verlag, Berlin, 2–2. DOI:http://dx.doi.org/10.1007/978-3-642-03237-0_2

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 27–37. DOI:http://dx.doi.org/10.1145/268998.266641

Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 11–21. DOI:http://dx.doi.org/10.1145/1375581.1375584

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT*

International Symposium on Foundations of Software Engineering (ESEC/FSE-13). ACM, New York, NY, 263–272. DOI:http://dx.doi.org/10.1145/1081706.1081750

Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I use the wrong definition? DeFuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, New York, NY, 160–174. DOI:http://dx.doi.org/10.1145/1869459.1869474

John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*. ACM, New York, NY, 158–167.

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*. Springer-Verlag, Berlin, 134–153.

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing production run failures at the user's site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 131–144. DOI:http://dx.doi.org/10.1145/1294261.1294275

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot—A java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. IBM Press, 13.

Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. 2005. A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE Computer Society, Washington, DC, 217–228. DOI:http://dx.doi.org/10.1109/CGO.2005.3

Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. ACM, New York, NY, 97–107. DOI:http://dx.doi.org/10.1145/1007512.1007526

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE'07)*. ACM, New York, NY, 205–214. DOI:http://dx.doi.org/10.1145/1287624.1287654

Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. 2011. ORDER: Object centric deterministic replay for java. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, 30–30.

Cristian Zamfir and George Candea. 2010a. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 321–334. DOI:http://dx.doi.org/10.1145/1755913.1755946

Cristian Zamfir and George Candea. 2010b. Low-overhead bug fingerprinting for fast debugging. In *Proceedings of the 1st International Conference on Runtime Verification (RV'10)*. Springer-Verlag, Berlin, 460–468.

Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting concurrency bugs through sequential errors. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, 251–264. DOI:http://dx.doi.org/10.1145/1950365.1950395

Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 179–192. DOI:http://dx.doi.org/10.1145/1736020.1736041

Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2012. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 892–902.

Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. 2014. MIMIC: Locating and understanding bugs by analyzing mimicked executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. ACM, New York, NY, 815–826. DOI:http://dx.doi.org/10.1145/2642937.2643014

Received June 2015; revised December 2015; accepted January 2016