Production-guided Concurrency Debugging

Nuno Machado
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal
nuno.machado@tecnico.ulisboa.pt

Brandon Lucia
Carnegie Mellon University, USA
blucia@ece.cmu.edu

Luís Rodrigues
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal
ler@tecnico.ulisboa.pt

Abstract

Concurrency bugs that stem from schedule-dependent branches are hard to understand and debug, because their root causes imply not only different event orderings, but also changes in the control-flow between failing and non-failing executions. We present Cortex: a system that helps exposing and understanding concurrency bugs that result from schedule-dependent branches, without relying on information from failing executions. Cortex preemptively exposes failing executions by perturbing the order of events and control-flow behavior in non-failing schedules from production runs of a program. By leveraging this information from production runs, Cortex synthesizes executions to guide the search for failing schedules. Production-guided search helps cope with the large execution search space by targeting failing executions that are similar to observed non-failing executions. Evaluation on popular benchmarks shows that Cortex is able to expose failing schedules with only a few perturbations to non-failing executions, and takes a practical amount of time.

1. Introduction

Concurrent programming has hit the mainstream, because it enables software to take advantage of parallelism in pervasive multicore computer architectures. Unfortunately, expressing concurrency in multi-threaded code is more challenging than writing sequential code. The reason is that multi-threaded programs often permit many different *schedules* of operations from different threads. Hence, a program's outcome may vary from run to run, depending on the executed schedule. Many schedules are correct, but some *failing schedules* result in misbehavior, like a crash or data corruption.

A large body of research has focused on exposing failing schedules and exhaustively testing for failures [12, 15, 17, 47]. To find a failing schedule is a challenging problem: the subset of thread orderings that lead to the failure often corresponds to a tiny portion of the space of possible execution schedules and those few failing schedules may manifest rarely. Furthermore, the variation in the schedule in a failing execution may cause a variation in the execution's data flow, and subsequently, its control-flow. Such *schedule-dependent branches* further complicate the task of exposing failing schedules, because one has to explore not only the space of possible thread schedules for a given execution path, but also the space of different execution paths. Since this huge search space makes complete exhaustive testing infeasible, some failures will likely manifest in deployment.

This work aims at exposing failures that may include schedule-dependent branches, without ever needing to observe a failing execution. We leverage the observation that a failing schedule typically deviates in only a few critical ways from a non-failing schedule [33]. Our main insight is to expose *new* failing schedules by perturbing the order of events and certain branch outcomes in a non-failing schedule. We further leverage abundant *production runs* of a program on deployed systems to guide our search of the enormous space of possible execution schedules [5]: our *production-guided search* for a failing schedule targets schedules very similar to a non-failing schedule observed in production.

We present Cortex[1], a system that helps exposing and understanding concurrency bugs using traces from normal, non-failing production executions. Figure 3 depicts an overview of our system. Cortex starts by collecting a set of per-thread path profiles from one or more production runs. Each profile is used to guide a symbolic execution of the program producing a symbolic event trace for each thread that is compatible with the original execution's control-flow. Cortex combines an execution's per-thread symbolic traces to implement *production-guided* search for a new, failing execution – one that may depend on *both* schedule and path conditions.

Cortex's production-guided search is a novel approach to selecting a path and schedule. Starting from the computed symbolic execution, Cortex systematically reorders events in the schedule and inverts the outcome of certain branches, with a preference for executions that are most similar to the original. Cortex determines if a perturbed execution is feasible using a constraint system for a *Satisfiability Modulo Theories* (SMT) solver. The constraint system encodes synchronization, data-flow, event ordering, and the occurrence of a failure. If the SMT formulation is satisfiable, the execution that Cortex generated is feasible and the system reports the new failure. If the SMT formulation is infeasible, Cortex moves on to a different perturbation of the execution's schedule and branching behavior. Cortex favors executions that vary only slightly from the original, observed execution, putting its focus on failures that very nearly manifested in a previous execution.

In this paper, we consider failures to be violations of assertions in the code. We argue that it is common for developers to ship code with assertions. For instance, Google pervasively uses tracing and assertions throughout live, production datacenter code via Dapper [45]. Also, recent work showed that invariants can often be derived automatically [24], which broadens the applicability of Cortex.

[1] We have named our system Cortex after the cerebral cortex, which is a part of the human brain that receives and processes information from neurons to control several functions of the human body. Likewise, our system leverages information from multiple production runs to expose concurrency bugs.

PPoPP '16, March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2851141.2851149

In addition to exposing failing schedules, Cortex is also able to isolate the failure's *root cause*. The root cause of a failure is the minimum sequence of events in the schedule that cause the program to fail. A failing schedule may contain many events and the failure's root cause could be anywhere in the schedule, which makes debugging a complex task. Failures resulting from schedule-dependent branches further exacerbate this problem, because the programmer must reason about different events in a failing execution and in a non-failing one.

Cortex leverages production-guided search to extend previous work on *differential schedule projections* [33] and compute *differential path-schedule projections* (DPSPs). DPSPs zero in on the root cause of failures that stem from schedule-dependent branches by reporting the differences between a failing and a non-failing schedule, including variations in their event orderings, data-flow behavior, and control-flow decisions.

Our evaluation in Section 6 shows that Cortex is able to find failing schedules in concurrent programs by perturbing very few branch conditions. Moreover, we show that Cortex's production-guided search reduces the number of attempts to expose concurrency bugs by up to three orders of magnitude with respect to previous state-of-the-art concurrency testing techniques [12, 17].

In summary, this paper makes the following contributions:

i) A cooperative scheme to collect and analyze thread path-profiles from production runs.

ii) A novel, *production-guided* approach to exposing path and schedule dependent failures by exploring variations in schedule *and* control-flow behavior in non-failing executions.

iii) A technique to synthesize new executions similar to observed ones leveraging traces collected from production, as well as symbolic execution.

iv) An implementation of Cortex for Java and an evaluation, with widely used benchmarks and real-world applications, showing that Cortex is efficient and effective for exposing and debugging hard concurrency bugs.

The rest of the paper is organized as follows. Section 2 overviews the background concepts most related to our work. Section 3 describes the Cortex system, namely its architecture and the production-guided search used to find failing schedules. Section 4 provides a concrete example that illustrates how Cortex employs the production-guided search to expose a concurrency bug that depends on schedule-sensitive branches. Section 5 discusses the implementation details. Section 6 presents the experimental evaluation results and discuss the main findings. Finally, Section 7 reviews the related work.

2. Motivation and Background

Cortex exposes new concurrency bugs and helps their diagnosis. This section overviews concurrency bugs and debugging, as well as techniques from prior work that form the foundation of Cortex.

2.1 Concurrency Bugs

Concurrency bugs are errors in code that permit operations from different threads to execute in an order that causes the program to fail — *concurrency bugs permit failing execution schedules*. Concurrency bugs include omitted and misused synchronization and inter-thread communication. For example, consider the multithreaded program in Figure 1. This program has two threads (T1 and T2), which access a shared variable x. T1 increments the value of x (which is initially set to 0) and then checks whether this value is greater than 0. In turn, T2 simply writes 0 to x.

```
          (initially x = 0)
        T1                    T2
 1: x++                3: x = 0
 2: assert(x > 0)
```

Figure 1: Example of a multithreaded program with a schedule-dependent bug.

This program has two possible outcomes: it either validates or violates the assertion at line 2, depending on the order in which threads execute their operations. The program fails for the schedule 1-3-2 and ends correctly for the schedules 3-1-2 and 1-2-3. The example is an *atomicity violation*, because the block of operations in T1 should execute atomically, without being interleaved by operations from T2, but the code fails to enforce the atomicity. Atomicity violations and other types of concurrency bugs (*e.g.* ordering violations and data-races) have been studied extensively in the literature [9, 14, 27–31, 38, 43, 53, 54] and we defer to prior work for a more thorough background on concurrency bug types.

Regardless of the type, some concurrency bugs are strictly *schedule-dependent*. The error in Figure 1 is an example of a strictly schedule-dependent bug: threads in both the failing and non-failing schedules execute the same sequence of instructions, but the schedules differ in the threads' operations interleaving.

Unlike the bug in Figure 1, not all concurrency bugs are strictly schedule-dependent. Some concurrency bugs are *path and schedule dependent*, instead, like the example in Figure 2. In the example, T1 and T2 access four shared variables (x, y, w, and z). The program fails when it executes the schedule 8-1-2-3-4-9-10-5-6-7, which causes the value of x at line 7 to be 0 and violate the assertion. All non-failing schedules for this program exhibit a different *control-flow path* than the failing schedule, because the code must not execute line 6, to guarantee that $x > 0$ at line 7. The key distinction between this example and the example in Figure 1 is that the failing execution requires a variation from the correct execution in both the order of events (*i.e.*, the *schedule*) and in the control-flow path executed. The next section discusses the challenges of path and schedule dependent bugs.

```
      (initially x = y = w = z = 0)
        T1                    T2
 1: if(z > 0)           8: z = 1
 2:    w++              9: if(w > 0)
 3: x = 1              10:    y = 0
 4: y = 1
 5: if(y == 0)
 6:    x--
 7: assert(x > 0)
```

Figure 2: Example of a multithreaded program with a path and schedule dependent bug.

2.2 Challenges of Path and Schedule Dependent Bugs

Testing for and debugging path and schedule dependent bugs is more complex and challenging than for strictly schedule-dependent bugs. The reason for the difference is that path and schedule dependent bugs require searching not only for an inter-thread operation order, but also for a new execution path that is compatible with such a failing schedule.

Testing. Stateless model checking systems like con2colic testing [11] and MCR [17] leverage SMT constraint solving to expose concurrency bugs. By encoding the possible thread schedules for a given execution path as a constraint system, these systems are able to check properties and search for failures in a set of schedules, for a given single execution path. Other *systematic concurrency testing* techniques systematically exercise many different schedules for the same control-flow path [16, 34]. Systematic schedule search works for strictly schedule-dependent bugs, but path and schedule dependent bugs require a tool to simultaneously explore all control-flow paths to find a path and schedule that leads to the failure. This requirement illustrates an important challenge: the space of paths and schedules *explodes* with an execution's length and quickly becomes unwieldy and infeasible to search. In this work, we address search space explosion using the local, production-guided search technique that we describe in Section 3.3.

Debugging. Debugging tools like CLAP [20] and Symbiosis [33] use symbolic execution and SMT constraint solving for automatic debugging. Both systems use per-thread path profiles from a *concrete failing execution* to generate and replay a failing schedule. While CLAP is only focused on reproducing failures, Symbiosis is able to diagnose them as well, by systematically reordering operations in a failing schedule to produce an alternate schedule that does not trigger the failure. Symbiosis uses a differential analysis of the alternate and the original schedule to isolate a failure's cause and produce a *differential schedule projection* [33].

Symbiosis and CLAP do not handle path and schedule dependent bugs: CLAP only computes failing schedules and Symbiosis' alternate schedules are required to adhere to the original control-flow of the original execution. Furthermore, both Symbiosis and CLAP need a trace from a failing execution to work, which may be hard to obtain. For instance, we ran the program in Figure 2 10,000 times and it did not fail a single time.

In this work, Cortex couples its production-guided search with symbolic execution and SMT solving ideas from Symbiosis and CLAP to address schedule and path dependent bugs without the need to observe a failing execution. Section 7 provides a more thorough comparison between Cortex and Symbiosis.

2.3 Computing Schedules with Symbolic Execution and Constraint Solving

Cortex leverages the multithreaded trace generation technique developed in CLAP [20] and refined in Symbiosis [33]. The technique uses concrete, per-thread path profiles to guide a symbolic execution of the program and generate per-thread symbolic traces (see Section 3.2). The technique then builds an SMT constraint formulation that is based on the per-thread symbolic traces. When solved by an off-the-shelf SMT solver, the formulation yields a failing, multi-threaded schedule. We defer a full discussion of this formulation to its original work [20], but provide background here.

The constraint formulation has two kinds of variables: *value variables*, which represent symbolic values returned by read operations, and *order variables*, which represent the order of operations in a schedule. The constraint system, Φ_{fail}, is a conjunction of five sub-formulae:

$$\Phi_{fail} = \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \neg\phi_{assert}$$

Briefly, ϕ_{path} encodes the path conditions corresponding to the path executed by each thread; ϕ_{sync} encodes inter-thread ordering imposed by synchronization; ϕ_{rw} encodes inter-thread ordering implied by accesses to shared memory; ϕ_{mo} encodes possible operation reorderings permitted by the memory consistency model; and $\neg\phi_{assert}$ encodes the failure constraint, which corresponds to an assertion failure.

Symbiosis [33] observed that a similar constraint formulation, Φ_{ok}, yields a non-failing schedule, without the failure condition negated:

$$\Phi_{ok} = \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{assert}$$

Therefore, for a given execution control-flow, the constraint system can be used to obtain an execution schedule that either fails or ends successfully, depending on whether the failure condition is satisfied or not, respectively. In the following, we describe each sub-formula in more detail.

Path Constraint ϕ_{path} is a conjunction of all threads' path conditions (*i.e.*, branch outcomes), recorded during symbolic execution [4, 51]. For instance, a possible path constraint for an execution of the program in Figure 2 would be $[z > 0] \wedge [\neg(y{=}{=}0)]$ for T1 and $[w > 0]$ for T2.

Synchronization Constraints. ϕ_{sync} is divided into *partial order constraints* and *locking constraints*.

The former encode the happens-before relation [25] between ordering synchronization operations (*e.g.*, signal, wait, join, fork). For instance, the constraints state that *i)* one thread's *start*/*join* event happens after another thread's *fork*/*exit* event. Locking constraints encode mutual exclusion of code protected by a lock. These constraints match a thread's *unlock* operations to a preceding *unlock* in that thread.

Read-Write Constraints. ϕ_{rw} encodes the ordering and result of shared memory read and write operations. The read-write constraints express the fact that a read from a variable returns the symbolic value written by the last write to that variable.

For example, in Figure 2, if T1 reads the value 1 for the variable z at line 1, then the most recent write to z must be the one at line 8 by T2 and, consequently, line 8 must execute before line 1.

Memory Order Constraints. ϕ_{mo} determines the order in which operations occur in a specific thread. In Cortex, we consider that operations execute in program order, *i.e.* following a sequencial consistent memory model. However, it is possible to express more relaxed memory consistency models using slightly different constraints [20].

Failure Constraint. ϕ_{assert} corresponds to the condition that, if unsatisfied, indicates that an execution failed. For a failing execution, Φ_{fail}, the failure condition must be violated (*i.e.*, $\neg\phi_{assert}$ must hold). On the other hand, for a correct execution Φ_{ok}, ϕ_{assert} must be true. In Figure 2, $\neg\phi_{assert}$ corresponds to $[x{\leq}0]$ and ϕ_{assert} corresponds to $[x{>}0]$.

3. Cortex

Cortex is an automated system for exposing and debugging path and schedule dependent failures in multithreaded programs. In contrast with other systems, Cortex does not need to observe a failed execution to isolate a failure. Instead, Cortex starts from a concrete, non-failing execution and explores alternative executions with only minor variations in their schedule and path from the non-failing schedule. Using an initial, non-failing execution from production turns Cortex's execution space exploration into a *production-guided search* for new failures. The exposed failures represent behavior that nearly happened in the observed execution, and is, thus, more likely to happen in some future execution. Cortex summarizes only the differences between the exposed failing execution and the original non-failing execution to clearly isolate the *root cause* of the failure to the developer.

Cortex operates in four main steps: *static analysis*, *trace collection*, *production-guided search*, and *root cause isolation*. These steps are illustrated in Figure 3 and described in the following sections.

3.1 Static Analysis

Cortex starts by performing a static program analysis with two goals. The first goal is to instrument the beginning of each basic

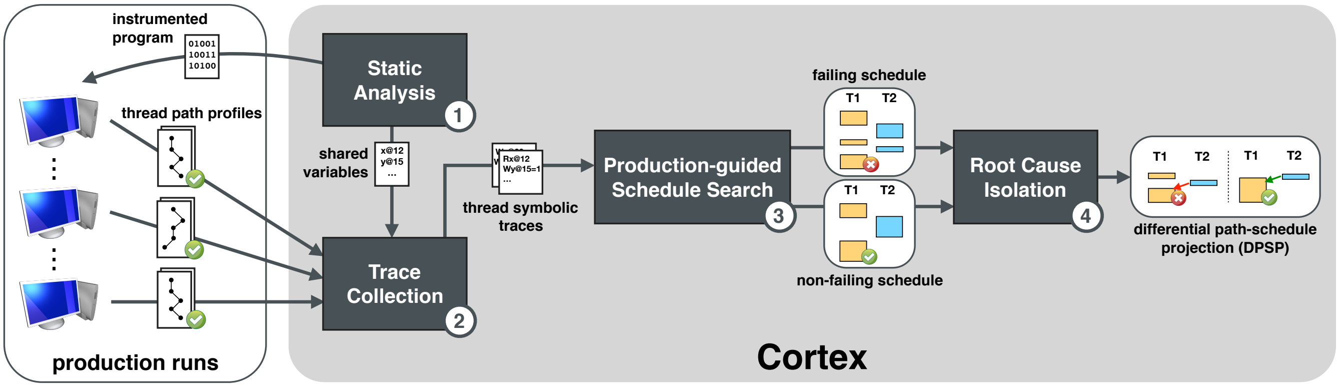

Figure 3: **Overview of Cortex.** 1) Cortex starts by instrumenting the program in order to allow capturing per-thread path profiles at runtime. 2) Cortex uses the path profiles, collected from multiple production runs, to guide a symbolic execution of the program and produce per-thread symbolic event traces that adhere to the original executions' control-flow. 3) Cortex leverages symbolic traces and SMT constraint solving to conduct a production-guided search for a new failing execution that may depend on both schedule and path conditions. 4) Whenever Cortex successfully uncovers a new failing schedule, it computes a differential path-schedule projection (using the uncovered failing schedule and a non-failing schedule) to isolate the underlying bug.

block in the program to trace the control-flow path followed in a concrete execution. The second goal is to identify shared variables. Non-private (*i.e.*, shared) variables and local variables derived from those variables are marked as symbolic. Marking shared variables as symbolic is a pre-requisite to Cortex's symbolic trace collection and generation mechanism (described next).

3.2 Trace Collection

Given the infeasibility of exhaustively exploring all possible executions of a program, Cortex narrows the exploration to favor those paths that are most like those observed in production.

Concrete Trace Collection. The version of the program instrumented by Cortex collects per-thread *path profiles* [20, 33]. A path-profile is a sequence of basic blocks (*i.e.*, control-flow outcomes) that was followed by the thread at runtime.

Note that traces collected at this stage do not have any information concerning thread scheduling, nor regarding which events were executed during the production run. The reason is because recording the precise thread interleaving incurs heavy runtime overhead [19]. Hence, as previous work [20], we opt for a more lightweight logging approach to capture runtime information.

Production runs with the same control-flow output identical traces. However, paths with different control-flow may execute in production. Cortex can collect a variety of paths from multiple, different executions, potentially from distinct machines and execution environments. This form of cooperative path collection enables Cortex to leverage the execution diversity in multiple deployments to gather a representative collection of traces.

Symbolic Trace Generation. Cortex uses symbolic execution to generate per-thread, symbolic traces from the collected, per-thread, concrete traces. Cortex augments the collected, per-thread path traces *offline* with information about shared variables from its static analysis. Cortex marks as symbolic all shared variables and local variables that depend on shared variables. These symbolic annotations mean that later, when Cortex does symbolic execution, reads and writes to these variables manipulate symbolic expressions, rather than concrete values.

Cortex performs a symbolic execution of the program, guiding each thread's symbolic execution by its path trace. In other words, the symbolic execution proceeds solely across the execution path indicated by its corresponding path profile. As a result, instead of exploring all control-flow paths, as in a conventional symbolic execution, the symbolic execution in Cortex only follows branches taken in the original run. Cortex, thus generates per-thread symbolic traces containing the same control-flow, synchronization, and shared memory accesses as the original, production run.

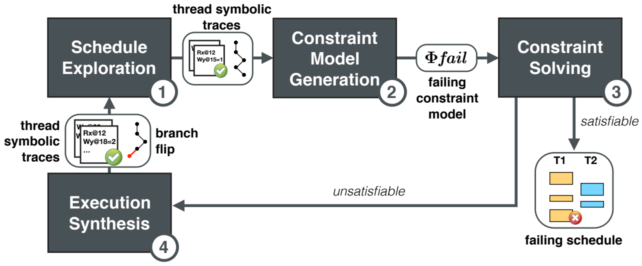

Figure 4: Detail of production-guided schedule search.

Cortex maintains a database of per-thread symbolic traces generated from any set of observed, per-thread concrete traces. Cortex organizes the traces in its database to facilitate its downstream search, using the executions' control-flow. Like symbolic execution tools [4, 48, 51], Cortex denotes a branch outcome as a binary value, with a 1 for branches taken and a 0 for branches not taken. An execution path is, thus, uniquely defined by a string of bits.

For example, consider a symbolic trace for thread T1 in Figure 2 with path id 10. This path id indicates that the thread followed an execution path corresponding to the conditions [z>0] and [¬(y==0)]. Note that the 0 in the path id means that the corresponding path condition evaluates false during the execution, hence the negation symbol in the condition [¬(y==0)].

3.3 Production-Guided Search

After gathering the path profiles from production runs and generating their corresponding per-thread symbolic traces, Cortex starts its search for alternate, failing executions. Figure 4 illustrates Cortex's search procedure.

Cortex's search is guided by production in two ways. The first way is *schedule exploration*, during which Cortex searches multithreaded executions compatible with the set of per-thread symbolic traces from some observed execution. If schedule exploration fails to surface any new failures, for any traces in Cortex's database, Cortex uses *execution synthesis* as a second form of production-guided search. Cortex synthesizes new multi-threaded executions by *modifying* the control-flow behavior in one or more of the threads' traces. Cortex then performs schedule exploration on the new synthesized execution.

3.3.1 Schedule Exploration

Given a combination of per-thread, symbolic path traces — from either a production run or a synthesized execution — Cortex checks if any interleaving of operations that make up those paths leads to a failure. First, Cortex selects an assertion from the trace at random. Next, Cortex builds a Φ_{fail} constraint model (from Section 2.3) with the symbolic information contained in the traces and the condition in the assertion.

Cortex uses an SMT solver to check the satisfiability of the generated constraints. If the model is satisfiable, then the solver outputs the failing schedule. If the constraints are unsatisfiable, then there is no schedule that leads to a failure of the selected assertion for the given control-flow trace. Cortex applies this schedule exploration procedure to each execution in its database, reporting newly exposed failures as they manifest.

Note that *only* performing schedule exploration on executions observed in production will only expose strictly schedule-dependent failures. However, Cortex is not limited to these failures only, because it goes beyond schedule exploration with its execution synthesis technique.

3.3.2 Execution Synthesis

Execution synthesis generates new per-thread control-flow traces corresponding to *entirely novel executions* by making small perturbations in the control-flow observed in production runs. By synthesizing new executions with control-flow variations, and then applying schedule exploration to those synthesized executions, Cortex can expose new failures that are schedule and path dependent.

Synthesizing executions presents two main challenges: *i)* how to decide which alternate execution to synthesize, and *ii)* how to obtain per-thread symbolic traces for the alternate execution to be synthesized. Cortex addresses the first challenge with a novel heuristic denoted *branch condition flipping*. The heuristic perturbs the original control-flow observed during some production run, generating one or more new per-thread traces. Cortex addresses the second challenge using a combination of its trace database and symbolic execution. If Cortex has already observed some execution in which a thread followed the perturbed control-flow path, Cortex uses the per-thread symbolic trace for that path that is in its database. If Cortex has not observed the perturbed path in some prior execution, Cortex synthesizes a new symbolic path trace by running a symbolic execution, guided by the perturbed control-flow trace.

Synthesizing a Control-flow Path. Cortex's synthesizes a new control-flow path by inverting path conditions on an existing path that are within a given distance from the selected assertion.

To identify the path conditions corresponding to the branches closest to the assertion, Cortex selects an execution from its database and generates a non-failing, multi-threaded schedule using the Φ_{ok} constraint model (from Section 2.3). Cortex examines the resulting schedule and selects the D branches closest to the assertion as candidates for inversion.

Cortex's path synthesis heuristic generates paths that are most similar to the original path trace first, generating new paths in order of deviation from the original. The first paths that the heuristic generates are ones with a single branch outcome flipped, and Cortex gives priority to paths in which the flipped branch is closer to the assertion. Next, the heuristic generates new paths with two branch flips, again, prioritizing new paths with lower total distance between branch flips and the assertion. Cortex's path synthesis heuristic continues considering complexes of increasingly many branch inversions up to the configurable threshold number D.

Figure 6a) illustrates Cortex's path synthesis heuristic. There are two threads, T1 and T2, and three branch conditions (Ⓐ and Ⓑ belong to T1, and Ⓒ belongs to T2). According to the non-failing schedule in Figure 6a), the closest branch to the assertion is Ⓐ, followed by Ⓑ and Ⓒ. The figure also shows that the path conditions in the threads' symbolic traces are 10 (*i.e.*, taken, not taken) and 1 (*i.e.*, taken), respectively for T1 and T2.

Figure 6b) depicts the first branch condition that Cortex attempts to flip, namely Ⓐ. As a result, Cortex generates a new control-flow path for T1 containing 11, while the trace for T2 remains the same. Later, the path synthesis heuristic may generate another control-flow path by flipping the outcome of multiple branches. Figure 6c) illustrates a case where Cortex simultaneously flips branches Ⓑ and Ⓒ, resulting in a path for T1 containing 00 and a path for T2 containing 0.

After synthesizing a new control-flow path, Cortex needs a new symbolic trace for the newly synthesized path. Cortex can either *find* an existing symbolic trace, or *synthesize* a new symbolic trace.

Finding a Symbolic Trace. The easiest way for Cortex to obtain a symbolic trace that is compatible with a newly synthesized control-flow path is to look for one in its database of traces collected from *any* prior, production execution. A compatible trace from the database must have an identical prefix of branch outcomes as the original, unperturbed trace, but must have the opposite outcome for the branch or branches flipped by the path synthesis heuristic.

When there is more than one compatible, symbolic trace in the database, Cortex considers each of them in turn, up to a maximum of N possible traces, and in ascending order of their path length. The tuple (D, N) allows tuning the search in terms of the number of different branches conditions flipped and the number of possible traces that are attempted for each branch flip. A high D means that Cortex flips path conditions far from the assertion, and a high N indicates that Cortex explores many paths with a common prefix.

Synthesizing a Symbolic Trace. When there is no trace in the database that matches a newly synthesized control-flow path, Cortex synthesizes a compatible trace using guided symbolic execution. Cortex uses the newly synthesized control-flow path to guide a symbolic execution of the thread up to, and including the flipped branch or branches. After reaching the flipped branch in the symbolic execution, Cortex has no information about which control-flow path to follow. Cortex allows the symbolic execution to run freely, exploring all paths, as in classical symbolic execution [4, 23, 51]. We heuristically stop the symbolic execution when it reaches the assertion or program exit along any path. We also stop symbolic execution after a configurable threshold timeout, to prevent the path explosion problem from hindering Cortex.

As an example, consider the scenario where Cortex has to synthesize the symbolic trace for T1 required in Figure 6b). Cortex would run the program symbolically, forcing T1 to take the branch 1 for the path condition Ⓑ, as well as for path condition Ⓐ. As T1's execution ends with the assertion right after Ⓐ, Cortex would output a symbolic trace for T1 that is compatible with the previously unobserved control-flow path 11.

3.4 Root Cause Isolation

Like prior systems on systematic concurrency testing [12, 17], Cortex is able to report a newly exposed failing schedule, but unlike prior systems, Cortex also reports a concise summary of the failure's root cause. To summarize a failure's root cause, Cortex computes and reports a *differential path-schedule projection* (DPSP). DPSPs are an extension of *differential schedule projections*, developed in Symbiosis [33]. Cortex computes DPSPs by analyzing an

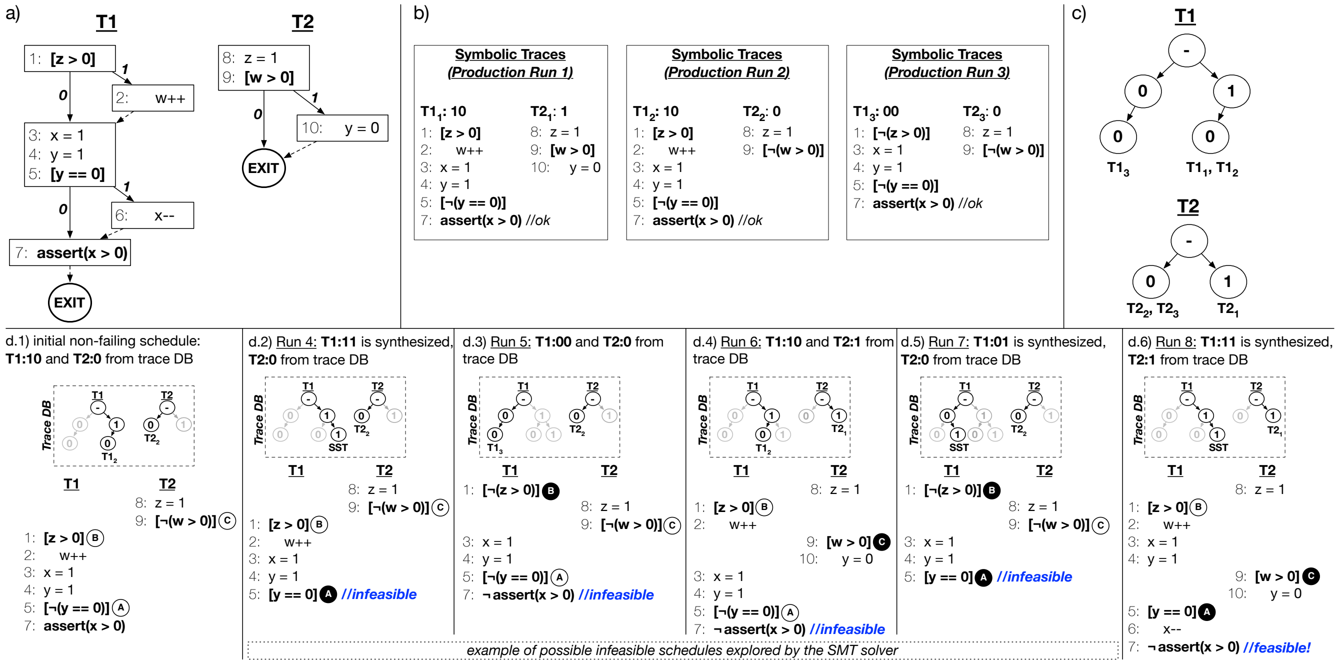

Figure 5: a) Schematic view of the program in Figure 2: boxes represent basic blocks, arrows depict conditional jumps (0 means *false* and 1 means *true*), dashed arrows depict unconditional jumps, round shapes represent program's exit points, and [z $>$ 0] represents a path condition; b) Per-thread symbolic traces path for three different correct production runs. T1_1:10 indicates that the trace for thread T1 from production run 1 has path id 10; c) Trace database, with per-thread path ids organized into prefix trees. The node label "-" indicates the root of the prefix tree. d) Production-guided schedule search employed by Cortex to find the failing schedule. SST stands for *synthesized symbolic trace*.

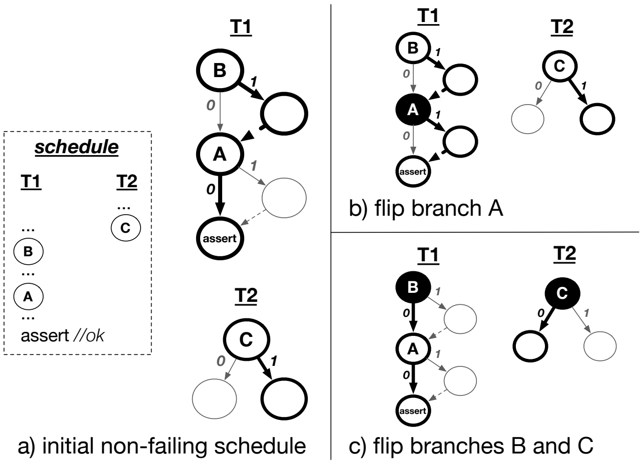

Figure 6: Branch condition flipping. Arrows and dashed arrows represent conditional and unconditional control-flow, respectively. Thicker arrows represent the execution path followed by the thread.

exposed failing execution and the original, non-failing execution that it was derived from. A DPSP reports the salient differences between the failing and non-failing schedule, including variation in their event orderings, data-flow behavior, and control-flow decisions. The key difference between DPSPs in this work and DSPs in prior work is that DSPs do not incorporate differences in control-flow between a failing and a non-failing execution, while, critically, DPSPs include those differences.

Cortex produces the DPSP by computing a "diff" of the failing schedule against the non-failing schedule. To compute a DPSP, Cortex first compares the traces and prunes a prefix of operations common to both the failing and non-failing schedule. Cortex then examines data-flow in both traces and reports only data-flow edges that exist in one trace, but not the other. Control-flow variations are highlighted as data-flow variations involving operations that only executed in one trace, but not the other. DPSPs are helpful for debugging, because they allow developers to see only a very small number of relevant operations and data movement events, rather than forcing them to pore over a full execution schedule. Furthermore, DPSPs illustrate the failure alongside a very similar, but non-failing execution. The side-by-side comparison helps understand the failure and aids in debugging.

4. Running Example

This section synthesizes the entire Cortex debugging workflow using a detailed running example. Figure 5 shows how Cortex automatically computes the root cause of the failure in Figure 2.

Static analysis. Cortex's static analysis identifies and instruments basic blocks and shared variables. Figure 5a shows the program's control-flow graph. In the example code, z, w, y, and x are marked as symbolic.

Symbolic trace collection. The program executes in production, potentially many times, and a path profile for each thread is collected from each execution. From the path profiles, Cortex produces symbolic traces. Using symbolic execution, Cortex identifies each branch condition evaluated by each thread (depicted in square brackets in Figure 5a) and symbolic execution follows those branches according to the path profile. The symbolic execution produces a corresponding symbolic trace file; Figure 5b shows the per-thread symbolic traces for three non-failing, production runs with different execution paths. For instance, T1_1:10 indicates that the trace for thread T1 of production run 1 followed the control-flow path 10 (i.e., taken, not taken). After producing the per-thread, symbolic traces, Cortex stores them in its trace database, depicted in Figure 5c as a prefix tree.

Production-guided search. Figures 5d.1-d.6 illustrate how Cortex uses production-guided search to expose a failing schedule from the non-failing, production schedules in its trace database.

First, Cortex tries to obtain a failing schedule by exploring the schedules that are compatible with per-thread traces from production runs that are in the trace database. Cortex applies its schedule exploration algorithm (see Section 3.3.1) to production runs 1, 2 and 3. In this example, there is no failing schedule that simply interleaves the per-thread traces from any execution. Instead, Cortex needs to explore alternate executions that it derives from the observed executions via execution synthesis.

Cortex begins its search by arbitrarily selecting production run 2, which includes traces T1$_2$ and T2$_2$. Using the traces from this execution, Cortex generates a non-failing schedule by calling out to the SMT solver (Figure 5d.1). Cortex examines the non-failing schedule to identify the branches that are closest to the assertion. In the example, these branches are Ⓐ, Ⓑ (from trace T1$_2$) and Ⓒ (from T2$_2$).

In Figure 5d.2, Cortex explores a different execution path by flipping the branch condition Ⓐ. The resulting path prefix is thus obtained by inverting the second bit in the path of trace T1$_2$, *i.e.*, by changing the path condition 10 to the path condition 11. Cortex checks its database for a symbolic trace for T1 with the path prefix 11, but, in this example, there is no such path in the database. Consequently, Cortex needs to synthesize a new symbolic trace for T1 with that prefix using symbolic trace synthesis.

Symbolic trace synthesis produces a symbolic trace T1:11. Cortex uses the generated trace, together with T2$_2$, to synthesize a new execution that we refer to as “run 4”. Cortex performs schedule exploration on run 4, checking for interleavings of its threads’ operations that lead to a failure. The solver, however, yields *unsatisfiable* when evaluating the constraint system that encodes schedule exploration. The execution is infeasible because there is no feasible data-flow that allows the value of y at line 5 to be 0.

To continue its search for a feasible schedule, Cortex again applies its path synthesis heuristic to generate a new control-flow path to explore (Figure 5d.3). The next branch to flip is Ⓑ from trace T1$_2$:10, which corresponds to the path 00. Cortex finds that its trace database for T1 already contains a trace with that path prefix (namely T1$_3$). Cortex uses the trace that it found to synthesize a new execution (“run 5”), containing the newly synthesized trace for T1 and trace T2$_2$. Cortex then performs schedule exploration on run 5, continuing its search for a feasible, failing alternate schedule.

Cortex proceeds according to this approach. When schedule exploration yields no failing schedule, Cortex synthesizes a new path, finds or synthesizes a new symbolic trace, creates a new execution, and re-applies schedule exploration. Figure 5d.4 and Figure 5d.5 show subsequent applications of the approach, which consist of runs 6 and 7, respectively. Note that, in Figure 5d.5, Cortex inverts the outcome of *two* branches, instead of a single branch because, at this point in its search, it has exhausted all options involving only a single branch inversion.

In Figure 5d.6, Cortex identifies a feasible, failing schedule for a newly synthesized execution that includes a synthesized symbolic trace for T1 (previously generated in run 4), and T2$_1$. The traces for T1 and T2 in the execution for which there is a failing schedule are the result of inverting the outcome of both branches Ⓐ and Ⓒ. Note that without Cortex’s unique ability to explore both schedule *and* path variations, this failure would not have been exposed.

Root cause isolation. With the failing and non-failing schedules that are the result of production-guided search, Cortex generates the DPSP depicted in Figure 7. On the left side is the non-failing schedule and on the right is the failing schedule. The DPSP does not show operations that the two schedules have in common, instead highlighting only the parts of the execution trace that are different. The DPSP illustrates (in the **bold** lines) which control-flow outcomes differ between the schedules. The arrows in the figure indicate data-flow edges that exist in one schedule, but not in the other. Together these properties of the DPSP show the root cause of the failure: the failure is attributable to a change in the order of operations in the schedule, the data-flow changes resulting from those ordering changes, and the control-flow changes stemming from the changes in data-flow.

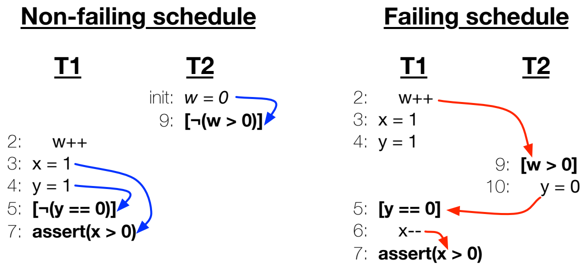

Figure 7: Differential path-schedule projection.

In particular, the example shows that the branch condition [w>0], which evaluates false in the non-failing schedule, becomes true in the failing schedule, because w at line 9 reads the value 1 (written by T1 at line 2) rather than the initial value 0. Consequently, T2 executes line 10 and sets y to 0, allowing the [y==0] to be true at line 5. In contrast, in the non-failing schedule, T1 takes the branch outcome corresponding to the condition ¬[y==0], because y at line 5 necessarily reads the value 1 written at line 4.

Finally, the DPSP shows that the assertion failure in the failing schedule is due to the value of x being decremented by T1 at line 6. Conversely, the execution ends successfully in the non-failing schedule because the read of x at line 7 returns the value 1, previously written at line 3.

Note that previous work in Symbiosis [33] simply reorders events in the failing schedule to obtain an alternate, non-failing schedule and produce a differential schedule projection. As such, Symbiosis would not be able to generate a DPSP like the one of Figure 7, because the failing schedule and the non-failing schedule for this case comprise not only sequences of different events, but also differing path conditions.

5. Implementation

We implemented a prototype of Cortex for Java programs. We use Soot [49] to perform the static analysis of Java bytecode, namely to inject probes at the beginning of each basic block that allow recording the path profile at runtime. Moreover, we leverage Soot’s *thread-local objects* (TLO) escape analysis to compute a sound over-approximation of the set of shared variables in Java programs. For each access on a shared variable we log an entry into a trace file containing the variable’s reference and the source code line. Cortex consults the trace file during symbolic execution to identify which operations should to treat symbolically.

Cortex’s production-guided schedule search and DPSP generation were implemented in around 1,200 lines of C and C++ code that extended the publicly available version of Symbiosis [33]. We extended Symbiosis to *i)* efficiently store symbolic traces from multiple production runs, *ii)* expose strictly schedule dependent, as well as path and schedule dependent concurrency bugs using traces from non-failing executions, and *iii)* perform symbolic trace synthesis during multiple path exploration.

Cortex organizes its database of per-thread path traces, represented as bit strings, into *tries* (*i.e., prefix trees*). Tries are typically used for string retrieval and contain one node for every common prefix of stored strings. In Cortex's implementation, if strings representing two different traces share a prefix of n bits, the corresponding executions followed the same path until the n^{th} branch decision. Cortex's use of a trie to store traces minimizes the storage required for large numbers of traces collected from production.

Cortex uses Java PathFinder (JPF) [51] for symbolic execution and Z3 [7] to solve SMT constraints. We modified Java PathFinder to integrate it with the other parts of Cortex. First, when generating symbolic traces for the production run per-thread path profiles, we ignore states that do not conform with execution path traced at runtime. This allows guiding the symbolic execution along the original paths only. Second, when synthesizing new symbolic traces, we force JPF to follow original path solely up to the branch condition flip point. After that, JPF switches to the traditional mode, where it explores all branches for each condition on symbolic variables, using a breadth-first search heuristic. In this mode, we also set a timeout to the exploration, in order to cope with path explosion.

Our Cortex prototype assumes that bugs in programs are expressed in the code as assertion invariants. Assuming that production software contains assertions is reasonable, as many major industrial environments use production assertions and tracing [45]. In our experiments, we added these assertions when they were not initially present. For cases where the error had the following form on the left, we inject the assertion as indicated on the right:

```
if(cond){              if(cond){
  //error                 assert(false)
}                         //error
else{                  }
  ...                  else{
}                         assert(true)
                          ...
                       }
```

Since JPF does not support arrays of symbolic length, we have also modified these cases in our experiments to have a constant size, without affecting the original buggy behavior of the program.

The Cortex prototype is publicly available at `http://github.com/nunomachado/cortex-tool`.

6. Evaluation

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

1. How efficient is Cortex in collecting symbolic traces from production runs? (§6.1)
2. How effective is Cortex's production-guided search in finding concurrency bugs? (§6.2 and §6.3)
3. How effective is Cortex in isolating the root cause of concurrency bugs? (§6.4)

We evaluated Cortex on the wide variety of multithreaded benchmarks shown in Table 1. These benchmarks have been used in prior work on concurrency debugging [10, 13, 17, 18]. We used 11 programs from the IBM ConTest benchmark suite [10]; *StringBuf*, a test driver of a bug in the Java JDK1.4 [17]; *ExMCR*, a micro-benchmark used by J. Huang *et al.* [17] to illustrate the benefits of MCR against other stateless model checking techniques. We have also tested with two real-world application bugs, namely *Pool* (which consists of a data race in Apache Commons Pool) and *Cache4j* (uncaught exception due to a data race). When presenting results, we sort test cases by "difficulty", *i.e*, benchmarks with fewer branches and smaller search spaces appear first in the tables.

Table 1: **Benchmarks and performance.** (LOC, #Threads, #Branches and #Shared stand for lines of code, number of threads, number of branches, and number of shared variable accesses in each benchmark).

Program	**LOC**	**#Threads**	**Profiling Overhead**	**Log Size**	**Symb. Exec.**	**#Branches**	**#Shared Events**
Account	373	5	18.1%	1KB	0.6s	1	244
Critical	76	3	17.3%	260B	0.59s	4	36
ExMCR	95	3	17.1%	170B	0.42s	4	56
PingPong	388	6	18.9%	226B	0.47s	5	66
Piper	280	5	18.6%	470B	1.11s	12	182
Airline	136	8	9.1%	252B	2.53s	15	77
Garage	554	7	6.7%	105KB	56.50s	22	284
BubbleSort	376	6	13.4%	1KB	0.59s	24	161
Manager	219	5	16.4%	1.4KB	0.79s	56	331
Loader	146	11	2.4%	4KB	0.91s	56	386
StringBuf	1339	3	19.9%	1KB	1.13s	65	331
TicketOrder	246	4	9.5%	892B	0.85s	69	354
BufWriter	272	5	20.4%	4.8KB	4.84s	89	1245
Pool	10K	3	2.5%	960B	1.4s	21	198
Cache4j (S)	2.3K	4	18.4%	3KB	1.02s	51	541
Cache4j (M)	2.3K	4	20%	15KB	2.01s	233	2364
Cache4j (L)	2.3K	4	21.7%	21KB	3.47s	309	3105

We modeled the data collection of a production environment by executing each program 100 times and ensuring that none of the 100 executions triggered the bug. From these production runs, we generated non-failing, symbolic traces and applied production-guided search to expose a failing schedule for each benchmark. For *Cache4j*, we have experimented with different workloads to assess the scalability of the constraint solving phase, as done in previous work [33]. Concretely, we re-ran this test case by varying the worker thread's update loop to have 1 (small), 5 (medium), and 10 (large) iterations.

The experiments were conducted on an 8-core, 3.5Ghz machine with 32GB of memory, running Ubuntu 10.04.4.

6.1 Cortex is Practical and Efficient

The most important result is that for all of the benchmarks that we considered, Cortex exposed a new failing execution based on a small handful of observed, non-failing schedules, and did so in a practical amount of time. Table 1 reports the time and storage overhead imposed by Cortex on production runs to capture path profiles, as well as the time required to compute symbolic trace collection. We report average time values across all executions (concrete and symbolic) for each benchmark.

Cortex's path profiling overhead ranges from from 2.4% in *Loader* to 21.7% in *Cache4j (L)*. The overhead is tolerable, even for production, and similar to other prior work in this area [20, 33]. Better software path profiling [3] or hardware support [50] are orthogonal techniques that would reduce this overhead.

Regarding space overhead, Cortex produces traces with sizes ranging from 170B in *ExMCR* to 105KB in *Garage*. The symbolic execution time is typically low as well: JPF produced a symbolic trace in less than one minute for all programs. The programs with larger path profiles are also the ones with more shared symbolic events (*e.g. BufWriter* and *Garage*). *Garage* has a long traces and symbolic executions time because it uses busy waiting.

6.2 Cortex Exposes Failures

Table 2 reports experimental results that allow assessing Cortex's efficacy in finding failing schedules. Columns 3 and 4 of the table together show that Cortex was able to find a failing schedule for all programs, including ones with failures dependent on both the path and schedule (" branch dependent "). We now characterize Cortex's ability to expose new failures.

Strictly schedule dependent bugs. Column 3 of Table 2 shows that schedule exploration alone works for only 8 out of the

Table 2: **Bug Finding Results.** Column 2 shows the number of different correct production runs observed; Column 3 marks bug found by schedule exploration only; Columns 4-8 provide details of production-guided search; Last column depicts the average time to solve the corresponding *satisfiable* SMT system.

Program	#Diff. Prod. Runs	Schedule Depend. Only	Branch Dependent					SMT Solving Time
				Tries	(D,N)	#Branches Flipped	#Synth. Traces	
Account	1	✓						29s
Critical	23		✓	6	(2,3)	2	4	<1s
ExMCR	1		✓	6	(4,1)	6	6	<1s
PingPong	39	✓						<1s
Piper	33		✓	1	(1,1)	1	1	1s
Airline	3	✓						<1s
Garage	2		✓	9	(3,4)	6	6	2s
BubbleSort	26		✓	4	(4,1)	4	4	<1s
Manager	39		✓	1	(1,1)	1	0	9s
Loader	1		✓	11	(11,1)	11	10	25s
StringBuf	12	✓						9s
TicketOrder	47		✓	1	(1,1)	1	0	1s
BufWriter	57	✓						2h56m
Pool	59		✓	17	(5,4)	15	8	1s
Cache4j (S)	11	✓						5s
Cache4j (M)	29	✓						1h30m
Cache4j (L)	37	✓						2h8m

17 benchmarks, namely *Account*, *PingPong*, *Airline*, *StringBuf*, *BufWriter*, and the three *Cache4j* scenarios. The reason schedule search alone is adequate for these benchmarks is that these eight cases include assertions of the form `if(cond){assert(false)} else{assert(true)}`. Cortex finds the failing schedule via schedule exploration alone because the failures are dependent only on strictly schedule dependent data flow to `cond`.

Efficiency of production-guided search. Column 4 in Table 2 shows that production-guided search finds a failing schedule for our path and schedule dependent bugs. Column 5 shows the number of branch outcome inversions Cortex performed to expose each failure. 3 out of 9 cases required inverting only the single closest branch to the assertion. This outcome supports our observation that failing executions are lurking in production, and that perturbing production executions is an effective search strategy for these otherwise elusive failures. The need for branch inversions, even in our production-guided search reinforces the fact that schedule exploration alone is insufficient.

Production run diversity. Collecting a diversity of production executions expedites Cortex's search for failures because it populates the trace database with traces, obviating the more costly execution synthesis step. Column 2 of Table 2 shows how many distinct non-failing executions Cortex observed during 100 runs of each benchmark. For all benchmarks except *BufWriter* and *Pool*, less than 50% of the collected executions are distinct. The data suggest that, even in small numbers of runs, executions are diverse and Cortex can leverage a large trace database in a large deployment.

Search parameters. The column labelled as (D, N) characterizes parameters used during Cortex's search for failures. Programs for which Cortex finds the failure with a single branch flip exhibit the pair (1,1) for (D, N). For those programs, Cortex exposed a bug by inverting the outcome of the single branch that was closest to the assertion.

In contrast, in *Critical*, *ExMCR*, *Garage*, *BubbleSort*, *Loader*, and *Pool* the optimal value for (D, N) varies significantly. For *Critical*, Cortex found the failing schedule after inverting two branch conditions (the *two* closest to the assertion) and performed schedule exploration using three different symbolic traces for each one of the two paths. For *ExMCR*, Cortex was able to find the failing execution in 6 attempts, but in this case it required 6 different combinations of branch inversions. Note that the number of attempts is actually greater than the product of the search parameters (D, N) for *ExMCR*, because Cortex needed to flip a combination of two branches simultaneously in order to trigger this failure[2]. *Garage* required fewer attempts than $D \times N$. The reason is that Cortex selected traces for some paths that included *redundant* execution paths. Cortex discarded the redundant paths and found a failing schedule using the 4^{th} trace for the 6^{th} combination of branch flips.

For *BubbleSort* and *Loader*, Cortex experimented with only one trace per branch inversion, but it searched through 4 and 11 branch inversions, respectively, to compute the failing schedule. *Pool*, in turn, was the program for which Cortex required more tries and flips of branch conditions to expose the failure. This is because the only combination of branch inversions that allowed finding the failing schedule corresponded to flipping simultaneously the 4^{th} and 5^{th} branches closest to the assertion.

In conclusion, these results show that search parameters (D, N) affect significantly the number of attempts that production-guided schedule search requires to expose the concurrency bug.

Solving Time. The last column of Table 2 reports the average amount of time that the SMT solver took to solve the constraint system (this value comprises only the case when the solver yielded *satisfiable*, because reporting *unsatisfiable* took at most 3 seconds for our test cases). The data shows that solving time is low for most cases, *i.e.*, a couple of seconds. The exception are benchmarks *BufWriter*, *Cache4j (M)*, and *Cache4j (L)*. Once more, this is due to the higher number of shared events in these programs. In particular, the solver took almost 3 hours for *BufWriter*, because the SMT constraint formulation for this program contains more than 920K read-write constraints and more than 6.5K locking constraints, which have a big impact in the solving time for this kind of constraint systems [20, 33].

6.3 Cortex Compares Favorably to Systematic Testing

Unlike Cortex, systematic testing techniques search by fully exploring the space of possible executions. We directly compared Cortex to two state-of-the-art systematic testing techniques, namely MCR [17] and iterative context bounding with dynamic partial order reduction (ICB-DPOR) [12]. Similarly to Cortex, MCR uses an SMT constraint-based approach to efficiently explore the space of possible schedules of a multithreaded execution in search for concurrency bugs. In particular, MCR starts from a concrete *seed interleaving* and builds a *maximal causal model* that allows checking correctness properties on all execution schedules equivalent to that seed interleaving. To further explore the state space, MCR iteratively generates new non-redundant schedules by enforcing read operations to return different values. MCR then uses the newly generated schedules as seed interleavings for subsequent iterations.

On the other hand, ICB-DPOR simply bounds the number of thread preemptions that can occur when systematically exercising different execution schedules, thus not accounting for redundant interleavings (*i.e.* interleavings that produce the same values for read operations).

Our goal is to show that Cortex examines *fewer executions* before exposing a failure than other approaches. We reproduce results for MCR and ICB-DPOR reported by Huang *et al.* [17] for the subset of benchmarks that have been used with all three systems. Table 3 reports the comparison.

The data show that, for most cases, Cortex searches orders of magnitude fewer executions than ICB-DPOR, and considerably fewer than MCR. The standout is *ExMCR*; *ExMCR* is a micro-benchmark that was designed to be *adversarial* to systematic con-

[2] We note that there are $2^D - 1$ different combinations of branch condition flips that can be attempted for a given search parameter D.

Table 3: **Comparison between Cortex and other systematic concurrency testing techniques.** Data for MCR and ICB-DPOR as reported by J. Huang *et al.* [17] ("*" indicates bugs that are schedule-dependent only). Shaded cells indicate the cases where Cortex outperforms the other systems.

Program	**#Attempts to find failing schedule**		
	Cortex	**MCR**	**ICB-DPOR**
Account*	1	2	20
ExMCR	6	46	3782
PingPong*	1	2	37
Airline*	1	9	19
BubbleSort	4	4	400
StringBuf*	1	2	10
Pool	17	3	6

currency testing systems. Cortex exposes the bug after searching just 6 executions, substantially outperforming both other systems.

The only benchmark where Cortex required more attempts to find the failing schedule than the other approaches was *Pool*. As mentioned before, for this program, Cortex was only able to trigger the failure after inverting the 4^{th} and 5^{th} branches at the same time. Hence, Cortex ended up spending time exploring combinations of branch flips that, despite being closer to the assertion, were ineffective to expose the failing schedule.

We believe the aforementioned scenario to be infrequent in practice, as shown by the outcomes of the other benchmarks. Therefore, we argue that the results in Table 3 further support our observation production-guided search is effective.

6.4 DPSPs are Concise and Informative

We computed DPSPs for all of our benchmarks, using observed (and synthesized) non-failing schedules and corresponding failing schedules exposed by Cortex. We evaluated DPSPs by comparing the number of data-flows and events in the DPSPs to those in full schedules. Table 4 summarizes our results.

The data show that DPSPs are simpler than full, failing schedules. DPSPs include only the salient differences between the failing and the correct executions, directing the developer's attention towards the most relevant events and the data-flows involved in the root cause of the failure. On average, Cortex produced DPSPs with 83% fewer data-flows and 50% fewer events than full schedules. Considering benchmarks with path and schedule dependent bugs alone, the average reduction values are 72% and 35%, respectively for data-flows and events. These results provide evidence that DPSPs are a useful asset for root cause diagnosis, not only for schedule-dependent only failures, but also for schedule and path dependent failures.

6.5 Discussion

The experimental results presented in the previous sections clearly demonstrate the benefits of Cortex in exposing and isolating path and schedule dependent bugs. Nevertheless, there are still a few challenges that need to be addressed in order to further improve Cortex's applicability and scalability. We enumerate these challenges below and discuss possible research lines to address them in the future.

Non-Assertion Bugs. Our current Cortex prototype assumes that failures manifest as assertion violations. Although assertions are commonly placed in code during development, concurrency bugs might not always be expressed as invariant failures.

One can address this issue by extending Cortex's constraint model to check for other type of concurrency bugs (*e.g.* data races [17] or deadlocks) in addition to assertion violations.

Input Non-determinism. In this paper, we assume that all production runs are captured with a fixed input. Consequently, we consider that the execution control-flow is only affected by thread interleavings. Exposing concurrency bugs considering variable input in addition to schedule non-determinism is substantially more challenging, because the search space now grows along two different dimensions: input and schedules.

Table 4: **DPSP conciseness.** Reduction achieved by Cortex in terms of number of data-flows and events with respect to full failing schedules ("*" indicates bugs that are schedule-dependent only).

Program	**#Data-flows Full**	**#Data-flows DPSP (%Red.)**	**#Events Full**	**#Events DPSP (%Red.)**
Account*	139	6 (↓**96%**)	244	58 (↓**76%**)
Critical	13	7 (↓**46%**)	36	13 (↓**64%**)
ExMCR	16	8 (↓**50%**)	56	39 (↓**30%**)
PingPong*	16	1 (↓**94%**)	66	5 (↓**92%**)
Piper	57	2 (↓**96%**)	182	51 (↓**72%**)
Airline*	29	3 (↓**90%**)	77	18 (↓**77%**)
Garage	139	26 (↓**81%**)	284	235 (↓**17%**)
BubbleSort	69	15 (↓**78%**)	161	144 (↓**11%**)
Manager	142	32 (↓**77%**)	331	220 (↓**34%**)
Loader	179	21 (↓**88%**)	386	281 (↓**27%**)
StringBuf*	115	1 (↓**99%**)	331	40 (↓**88%**)
TicketOrder	183	45 (↓**75%**)	354	290 (↓**18%**)
BufWriter*	745	95 (↓**87%**)	1245	1216 (↓**2%**)
Pool	73	34 (↓**53%**)	198	123 (↓**38%**)
Cache4j (S)*	211	1 (↓**99.5%**)	541	11 (↓**98%**)
Cache4j (M)*	862	3 (↓**99.7%**)	2364	1371 (↓**42%**)
Cache4j (L)*	1139	3 (↓**99.7%**)	3105	1094 (↓**65%**)

A possible way to address input non-determinism is to extend Cortex to record the input during production runs, in addition to threads' execution path. This way, symbolic traces could then be aggregated in subsets according to their execution path and input. Another approach is to mark as symbolic the variables that are affected by the inputs of the program, in order to force Cortex to explore the space of possible inputs during symbolic execution.

Long Executions. As shown in our experiments, when the execution has a large number of shared events, the SMT solver can take a long time to solve the constraint system. For long executions, this problem is exacerbated and it can become hard to expose a failing schedule in a reasonable amount of time.

To improve the scalability of Cortex's constraint solving phase, one could apply record-and-replay techniques [32, 37, 55] to capture lightweight information regarding the thread orderings observed at runtime. This data could then be used to prune the constraint model (by fixing some read-write linkages), without compromising the ability to expose failing schedules.

Alternatively, Cortex could leverage *interference abstractions* [46] to make the constraint analysis more tractable. Interference abstractions allow computing under- and over-approximations of thread interferences in a concurrent program. Moreover, interference abstractions can be gradually refined to reduce the space of possible schedules when checking for properties [46].

7. Related Work

A large body of prior work has studied debugging and testing of concurrent programs. In this section, we reiterate some of the solutions discussed in Section 2.2 and overview other prior efforts that are most related to Cortex.

Cortex vs Symbiosis. Symbiosis [33] is a system that helps developers to understand and diagnose concurrency failures by computing a differential schedule projection (DSP). DSPs are useful for debugging because they highlight the variations in data-flow between a failing schedule and a non-failing schedule, which allows isolating the bug's root cause.

As referred in Section 2.2, Cortex and Symbiosis leverage several similar techniques, namely guided symbolic execution, SMT constraint solving, and differential analysis to isolate bugs. However, we argue that these two systems are fundamentally different. A fundamental distinction of Cortex is that Symbiosis starts from a single failing execution and produces a single non-failing one. Symbiosis is, therefore, predicated on having evidence about the existence of a bug and its location. In contrast, Cortex *synthesizes* a failing execution from a set of *failure-free* executions. As such, Cortex must explore a much larger search space of executions. Furthermore, this space includes both branch and schedule variations: another key distinction from Symbiosis, which is not able to handle path variations.

Finally, as Symbiosis cannot isolate path and schedule dependent bugs like Cortex, we can say the DPSPs computed by Cortex are more broadly applicable than the DSPs produced by Symbiosis.

Cooperative tracing of production runs. Cortex's cooperative approach is inspired by prior work on cooperative tracing and debugging. CBI [26], CCI [21], and Gist [22] log events (e.g., branch frequency, return values) from multiple production runs and use statistical predictors to isolate the bug's root cause. LBR/LCR [2], in turn, uses on low-overhead hardware extensions to maintain a short-term log of hardware events that are useful for production run failure diagnosis. CoopREP [32] records partial logs from multiple user instances running a multithreaded program and combines that information to deterministically replay a concurrency error. Aviso [29] uses statistical analysis of production-run event traces, but with the orthogonal goal of avoiding failures, rather than exposing them.

Cortex benefits from information collected in production as well. However, Cortex has more immediately exposed failures than other cooperative techniques because it does not need the amount of data required by statistical approaches. Cortex has also the advantage of being able to synthesize new execution traces as necessary. Moreover, most other cooperative systems require first *observing* a failing execution, rather than exposing new ones.

Testing for concurrency bugs. *Systematic concurrency testing* (SCT) is one approach to testing multithreaded programs. The two most widely adopted SCT techniques are *partial order reduction* (POR) and *schedule bounding*. POR [15] reduces the number of schedules that need to be explored without false negatives, by exploring only one of each group of partial-order-equivalent set of executions. Dynamic POR techniques aim at improving the efficiency and effectiveness of POR by computing a persistent set [6], sleep set [12], and source set [1] during systematic search.

Schedule bounding techniques strive to limit the set of schedules examined during testing [8, 34, 39]. For instance, preemption bounding [34, 39] limits the number of preemptive context switches that are allowed in a schedule. Delay bounding [8], in turn, bounds the amount of times a schedule can deviate from the scheduling defined by a given deterministic scheduler.

CTrigger [36] attempts to reduce the interleaving space in exploration by focusing the testing on unserializable interleavings, which are interleavings that usually correspond to atomicity violations and have low probability of occurring outside a controlled environment. In turn, AtomFuzzer [35] relies on annotations provided by developers to dynamically check for atomicity violations in multithreaded programs. AtomFuzzer uses a random scheduler to choose an arbitrary thread to run at every program state, favoring interleavings that correspond to atomicity violation execution patterns. RaceFuzzer [44] also employs random testing, but with the goal of finding data races. Similarly to AtomFuzzer, RaceFuzzer combines a random scheduler with race detection techniques, in order to guide an execution schedule towards potential racing pairs of statements.

All of these techniques output a failing schedule and expose new concurrency bugs, but do not concisely summarize a failure's root cause, like Cortex.

Another approach to testing multithreaded programs is *test synthesis*. Test synthesis receives a suite of sequential tests as input, and analyzes the traces from these sequential executions in order to generate bug-inducing multithreaded tests. Omen [40], Narada [42], and Intruder [41] use this approach to automatically synthesize tests aimed at exposing deadlocks, races, and atomicity violations, respectively. Note, however, that test synthesis techniques still require multithreaded tests to be executed and analyzed with dynamic detectors [14, 35, 38] to find the concurrency bugs.

Symbolic execution and SMT constraints Symbolic execution and SMT constraint solving form the core of Cortex's schedule search. Prior work has also used a combination of symbolic execution and SMT constraint formulations to deterministically replay concurrency failures [20], test multithreaded programs [11, 17], find atomicity violations [52] and identify schedule-sensitive branches [18].

However, none of the techniques above use cooperative trace collection and production-guided search.

8. Conclusions and Future Work

We have presented Cortex, a system that is not only able to find concurrency bugs in programs, but also helps the programmer in identifying their root cause. For this, Cortex generates differential path-schedule projections (DPSPs) that capture the differences between non-failing and failing executions, even when threads follow different paths in each execution. These DPSPs have from 46% to 99% less data-flows than full failing executions, strongly simplifying the task of identifying the branches that are involved in the bug.

Contrary to most previous work, Cortex does not require a failure to be observed in production to avoid exploring the full space of possible executions. Instead, it is able to use non-failing executions from production runs as a starting point for exploration, generating synthetic executions that are likely to expose a bug (when it exists). Interestingly, with the benchmarks used in the paper, Cortex was able to generate failing executions after a few (more precisely, from just 1 to 15) cleverly guided branch flips.

In its current version, Cortex stops when a failing execution and the corresponding DPSP are produced. In future versions, we plan to search for, and compare, multiple failing executions, to enrich the information provided to the programmer.

Acknowledgements

We would like to thank our shepherd Murali Ramanathan and the anonymous reviewers for their invaluable feedback. This work was partially supported by Fundação para a Ciência e a Tecnologia (FCT), under project UID/CEC/50021/2013, and by a 2015 Google Faculty Research Award.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *POPL'14*, 2014.

[2] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *ASPLOS'14*, 2014.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4), July 1994.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, 2008.

[5] G. Candea. Exterminating bugs via collective information recycling. In *HotDep'11*, 2011.

[6] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. In *STTT'98*, 1998.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08/ETAPS'08*, 2008.

[8] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL'11*, 2011.

[9] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP'03*, 2003.

[10] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS'03*, 2003.

[11] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ESEC/FSE'13*, 2013.

[12] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL'05*, 2005.

[13] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI'03*, 2003.

[14] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI'08*, 2008.

[15] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[16] P. Godefroid. Model checking for programming languages using verisoft. In *POPL'97*, 1997.

[17] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI'15*, 2015.

[18] J. Huang and L. Rauchwerger. Finding schedule-sensitive branches. In *ESEC/FSE'15*, 2015.

[19] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent java programs. In *FSE'10*, 2010.

[20] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *PLDI'13*, 2013.

[21] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA'10*, 2010.

[22] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *SOSP'15*, 2015.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), July 1976.

[24] M. Kusano, A. Chattopadhyay, and C. Wang. Dynamic generation of likely invariants for multithreaded programs. In *ICSE '15*, 2015.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[26] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI'03*, 2003.

[27] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS XII*, 2006.

[28] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII*, 2008.

[29] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS'13*, 2013.

[30] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA'10*, 2010.

[31] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI'11*, 2011.

[32] N. Machado, P. Romano, and L. Rodrigues. Lightweight cooperative logging for fault replication in concurrent programs. In *DSN'12*, 2012.

[33] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. In *PLDI'15*, 2015.

[34] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08*, 2008.

[35] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE'08*, 2008.

[36] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS XIV*, 2009.

[37] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP'09*, 2009.

[38] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE '10*, 2010.

[39] S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR- TR-2007-12, Microsoft Research, 2007.

[40] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *OOPSLA '14*, 2014.

[41] M. Samak and M. K. Ramanathan. Synthesizing tests for detecting atomicity violations. In *ESEC/FSE 2015*, 2015.

[42] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *PLDI 2015*, 2015.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), Nov. 1997. ISSN 0734-2071.

[44] K. Sen. Race directed random testing of concurrent programs. In *PLDI'08*, 2008.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[46] N. Sinha and C. Wang. On interference abstractions. In *POPL '11*, 2011.

[47] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *PPoPP'14*, 2014.

[48] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *TAP'08*, 2008.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON'99*, 1999.

[50] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *CGO'05*, 2005.

[51] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA'04*, 2004.

[52] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS'10*, 2010.

[53] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS XV*, 2010.

[54] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: Detecting concurrency bugs through sequential errors. In *ASPLOS XVI*, 2011.

[55] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *ICSE'12*, 2012.