ConSeq: Detecting Concurrency Bugs through Sequential Errors

Wei Zhang[1] Junghee Lim[1] Ramya Olichandran[1] Joel Scherpelz[1] Guoliang Jin[1]
Shan Lu[1] Thomas Reps[1,2]

[1]Computer Sciences Department, University of Wisconsin–Madison [2]GrammaTech,Inc
{wzh,junghee,ramya,scherpel,aliang,shanlu,reps}@cs.wisc.edu

Abstract

Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses. Their effects propagate through data and control dependences until they cause software to crash, hang, produce incorrect output, etc. The lifecycle of a bug thus consists of three phases: (1) triggering, (2) propagation, and (3) failure.

Traditional techniques for detecting concurrency bugs mostly focus on phase (1)—i.e., on finding certain structural patterns of interleavings that are common triggers of concurrency bugs, such as data races. This paper explores a consequence-oriented approach to improving the accuracy and coverage of state-space search and bug detection. The proposed approach first statically identifies potential failure sites in a program binary (i.e., it first considers a phase (3) issue). It then uses static slicing to identify critical read instructions that are highly likely to affect potential failure sites through control and data dependences (phase (2)). Finally, it monitors a single (correct) execution of a concurrent program and identifies suspicious interleavings that could cause an incorrect state to arise at a critical read and then lead to a software failure (phase (1)).

ConSeq's backwards approach, (3)→(2)→(1), provides advantages in bug-detection coverage and accuracy but is challenging to carry out. ConSeq makes it feasible by exploiting the empirical observation that phases (2) and (3) usually are short and occur within one thread. Our evaluation on large, real-world C/C++ applications shows that ConSeq detects more bugs than traditional approaches and has a much lower false-positive rate.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing Tools

General Terms Languages, Reliability

Keywords Software testing, concurrency bugs

1. Introduction

1.1 Motivations

Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses. They exist widely (e.g., 20% of driver bugs examined in a previous study [50] are concurrency bugs) and are among the most difficult bugs to detect and diagnose. They have caused real-world disasters such as the Therac-25 medical accident [29] and the 2003 Northeast blackout [52]. Today, they are of increasing concern due to the pervasiveness of multi-core machines. Effective approaches to detecting concurrency bugs *before* software is released are sorely desired.

Reliability research for sequential software has made tremendous progress recently [9, 17, 41]. However, these successes have not translated to the world of concurrency software, because concurrency bugs are caused by a feature (namely, interleavings) not found in sequential programs.

Interleavings are not only complicated to reason about, but they also dramatically increase the state space of software. For large real-world applications, each input easily maps to billions of execution interleavings, and a concurrency bug may only be exposed by one specific interleaving. How to analyze this huge space *selectively* and expose hidden bugs is an open problem for static analysis, model checking, and software testing.

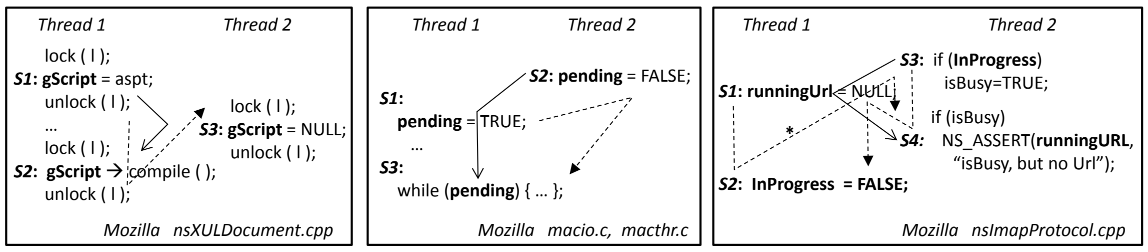
To address this challenge, existing techniques for detecting concurrency bugs are mostly guided by certain structural patterns of interleavings that most frequently cause concurrency bugs. These patterns include data races (conflicting accesses to a shared variable) [12, 19, 42, 51, 64], simple atomicity violations (unserializable interleavings of two small code regions) [33, 44, 56, 62], context-switch bounded interleavings [7, 27, 38, 39], etc. Although much progress has been made in this direction, fundamental limitations remain:

(1) False negatives (*are there other patterns of buggy interleavings?*). Many patterns have been proposed, while common real-world bugs that cannot be covered by traditional patterns keep popping up, such as multi-variable concurrency bugs [25, 31, 34] and order violations [32, 63], as shown in Figure 1.

(2) False positives (*are these interleavings truly harmful?*). Only about 2–10% of *true* data races are harmful [8, 40]. A similar rate holds for other interleaving patterns [44, 66]. Although innovative training [11, 31, 33] and testing [39, 44, 53] can mitigate this problem, lots of machine time and manual effort can be wasted on false positives.

(3) User unfriendliness. Interleavings are complicated to reason about. It is usually difficult for developers to judge whether a suspicious-looking interleaving snippet is truly a bug, and if so, how severe it is. Some commercial bug-detection tools choose not to flag concurrency errors solely because the reported buggy interleavings are too difficult to explain to developers [5]!

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03…$10.00



```
Thread 1                         Thread 2
    lock ( l );
S1: gScript = aspt;
    unlock ( l );
    …                                lock ( l );
    lock ( l );                  S3: gScript = NULL;
S2: gScript → compile ( );           unlock ( l );
    unlock ( l );
                                   Mozilla   nsXULDocument.cpp
```

```
Thread 1                  Thread 2
                          S2: pending = FALSE;
S1:
  pending = TRUE;
  …
S3:
  while (pending) { … };
                                Mozilla   macio.c,  macthr.c
```

```
Thread 1                          Thread 2
                                  S3: if (InProgress)
S1: runningUrl = NULL;                  isBusy=TRUE;
                                      if (isBusy)
                                  S4:   NS_ASSERT(runningURL,
                                          "isBusy, but no Url");
S2: InProgress = FALSE;
                                   Mozilla   nsImapProtocol.cpp
```

(a) This bug is not a race; it is an atomicity violation [33,56] ⟶ (b) This bug is not atomicity violation or race; it is an order violation [32] ⟶ (c) This is not a traditional bug; it is a multi-variable synchronization problem [31,35]

Figure 1. Bugs caused by various types of interleavings. (Solid and dotted arrows represent incorrect and correct interleavings, respectively. *: In (c), S1→S2→S3→S4 is a correct and feasible execution order, because `InProgress` could be set to TRUE and `runningURL` to a non-NULL URL string in between the execution of S2 and S3 by code not shown in the figure.)

1.2 Going beyond interleavings

ConSeq uses potential software failures to guide its search of the interleaving space. It uses a consequence-oriented approach to bug detection, which starts from the following observation [1]:

> *Concurrency and sequential bugs have drastically different causes but have mostly similar consequences.*

For example, the three bugs in Figure 1 all start with complicated interleavings, which cannot be detected by many existing detectors [31], and end up as common errors and failures similar to sequential bugs. The bug in Figure 1(a) is caused by an atomicity violation and cannot be detected by race detectors [33]; it causes a NULL-pointer dereference, and crashes in thread 1. The bug in Figure 1(b) is caused by an order violation: S2 could unexpectedly execute before S1, in which case the `FALSE` value of `pending` would be overwritten too early. This problem cannot be fixed by locks, and cannot be correctly detected by atomicity-violation or race detectors [32]. It causes an infinite loop in thread 1. The bug in Figure 1(c) has a different cause from the above two bugs. Since S1→S2→S3→S4 and S3→S4→S1→S2 are both correct interleavings, the specific order between S1's and S4's accesses to `runningUrl` is not responsible for the software failure. Similarly, the specific order between S2 and S3 does not matter. It is the unsynchronized accesses to *two* correlated shared variables that lead to an assertion failure. Sophisticated multi-variable concurrency-bug detectors [31] were designed to detect this type of bug.

To better understand the above observation, Figure 2 depicts the three-phase life cycle of a concurrency bug [28, 47]. After being *triggered* by an incorrect execution order across multiple threads, a concurrency bug usually *propagates* in *one* thread through a short data/control-dependence chain, similar to one for a sequential bug [21]. The erroneous internal state is propagated until an externally visible failure occurs. At the end, concurrency and sequential bugs are almost indistinguishable: no matter what the cause, a crash is often preceded by a thread touching an invalid memory location or violating an assertion; a hung thread is often caused by an infinite loop; incorrect outputs are emitted by one thread, etc. The only major class of concurrency errors that have substantially different characteristics is deadlocks, which fortunately have been thoroughly studied [24, 57].

Given the above observation and the trouble presented by the enormous space of interleavings, we naturally ask, "How can we leverage the sequential aspects of concurrency bugs? Can we reverse the bug-propagation process and start bug detection from the potential points of failure?" This approach, if doable, has the potential to leverage common error-propagation patterns shared between concurrency and sequential bugs:

- In terms of false positives, the questions 'is this a bug?' and 'how harmful is this bug?' are easier to answer with this technique, because causing a failure is the criterion used for deciding whether or not to report a bug.
- In terms of false negatives, it can provide a nice complement to existing cause/interleaving-oriented detectors, because what ConSeq detects is not limited to any specific interleaving pattern.
- ConSeq's reports are likely to be more accessible to developers who are familiar with the effects/symptoms of sequential bugs. Moreover, developers can now contribute to the bug-detection process by writing per-thread consistency checks, and by specifying which potential failures are worth more attention.

The consequence-oriented approach to bug detection presents many challenges. Backward analysis might be straightforward for failure replay and diagnosis [1, 15, 46, 58, 65] when the failure has already occurred. However, it is much more difficult for bug-detection and testing, where we need to identify *potential* points of bugs and failures. The problem we face is more similar to proving whether a specific property in a concurrent program could be violated. This is known to be a hard problem in software verification, even when attempted on small programs, and explains why little work has been done in this direction. In this paper, we show that it is possible to apply such techniques to large, real-world C/C++ programs (with millions of lines of source code and object files tens of megabytes long).

1.3 Contributions

This paper proposes ConSeq, a consequence-oriented backward-analysis framework to detect concurrency bugs. ConSeq starts its detection by identifying potential failure sites in the program. It then uses static analysis and run-time information collected from correct runs to decide whether these failures could occur under other interleavings. It can automatically and accurately detect concurrency bugs—with various types of root causes—before they occur, including atomicity violations, order violations, anti-atomicity violations, multi-variable synchronization problems, and others.

To carry out consequence-oriented bug-detection, ConSeq addresses two major challenges:

Challenge I: how to automatically identify *potential* error sites *before* an error occurs? Naive solutions will identify either too many (theoretically, every instruction could generate or acquire an incorrect value) or too few (if limited to specific patterns) and cause accuracy and coverage problems.

To fully leverage the similarity between concurrency and sequential bugs' error/failure patterns, ConSeq statically identifies **five** types of potential error sites that cover almost all major types

[1] All the observations and characteristics mentioned here are discussed in more detail in Section 3.

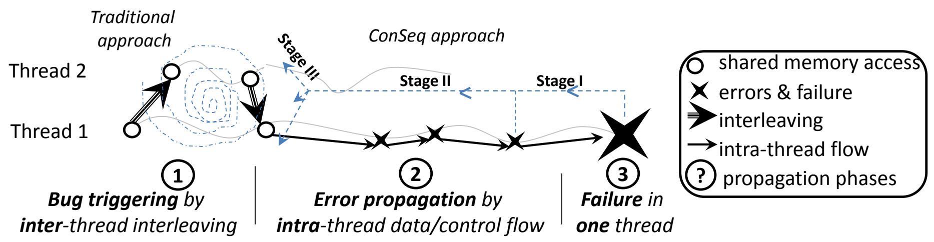


Figure 2. The common three-phase error-propagation process for most concurrency bugs.

Concurrency Bug Propagation & Characteristics		ConSeq Bug Detection
Phase 1: Triggering	●involving a small # of shared memory accesses	Step 3: Find and test suspect interleavings (trace-based synchronization analysis)
Phase 2: Propagation	●mostly within one thread ●mostly a short distance	Step 2: Identify error-inducing reads (static program slicing)
Phase 3: Errors & Failures	●mostly within one thread ●common error patterns	Step 1: Identify potential errors (thread-local static analysis)

Table 1. Observations about concurrency bugs and corresponding components of ConSeq.

of concurrency bug failures (**Stage I** of ConSeq as shown in Figure 2):[2] (1) calls to assertions in the software (for assertion crashes); (2) back-edge in loops (for infinite loop hangs); (3) calls to output functions (for incorrect functionality failures), (4) calls to error-message functions in the software (for various types of internal errors); (5) reads on global variables where important invariants likely hold according to Daikon [17], a tool for inferring program invariants (for miscellaneous errors and failures).

Challenge II: how to predict whether a potential error could occur in a *future interleaving*? Checking whether a specific property can be violated in concurrent programs is an NP-hard problem [23] and cannot be used for large real-world applications.

ConSeq addresses this challenge by exploiting two characteristics of concurrency bugs: first, the error-propagation distance is usually short in terms of data/control-dependence edges [21] (more validation in Section 3); second, the cause of a concurrency bug usually involves a specific ordering of just a few (two or three) shared memory accesses [7, 32].

Specifically, ConSeq first uses static slicing to identify shared memory reads that can impact each potential error site through a short chain of control/data dependences (**Stage II** of ConSeq in Figure 2). Then, after monitoring a correct run, ConSeq uses execution-trace analysis and perturbation-based interleaving testing to explore potential non-determinism around these reads (**Stage III** of ConSeq in Figure 2). Non-determinism that could affect the values acquired at those reads and lead to eventual errors and failures will be exposed effectively.

Overall, our work makes the following contributions:

(1) We report the results of a study of concurrency bugs' error-propagation characteristics. In a sample of 70 real-world concurrency bugs, we find that most non-deadlock concurrency bugs have short propagation distances (59 out of 70). In addition, failures in most concurrency bugs involve only one thread (66 out of 70).

(2) The work provides a new perspective on concurrency-bug detection and testing, which is to start from potential consequences and work backwards. It provides alternative interpretations for some concurrency bugs with complicated causes that are difficult to detect using traditional approaches, and sets up a nice connection with sequential bug-detection research, such as Daikon [17].

(3) We present a three-stage bug-detection framework that leverages characteristics from all three phases of the concurrency-bug propagation process (Table 1). This design separates the complexity of inter-thread interleaving analysis and intra-thread propagation analysis, and makes it easy to leverage advanced static-analysis techniques, such as slicing and loop analysis. Each stage of the framework can be easily extended. In particular, programmers can assist ConSeq by putting more consistency checks into their code, such as assertions and error messages.

(4) We implemented these ideas in a bug-detection tool that can analyze one correct run of a concurrent program and detect possible bugs that could occur in the future.

We evaluated ConSeq on 11 real-world concurrency bugs in seven widely used C/C++ open-source server and client applications. Results show that ConSeq is able to detect 10 out of 11 tested concurrency bugs, which cover a wide range of root causes, from simple races and single-variable atomicity-violations to order-violations, anti-atomicity violation bugs, multi-variable synchronization problems, etc. For comparison, we evaluated a race detector and an atomicity-violation detector and found that they could only detect 3 and 4 bugs, respectively. ConSeq detects these bugs with high accuracy: it has about one-tenth the false-positive rate of the race detector and the atomicity-violation detector.

We also found 2 new bugs in Aget, 2 new bugs in Click, and one output non-determinism in Cherokee, for which bugs have not been previously reported. We found a known infinite-loop bug in a version of MySQL for which the bug had not been previously reported. Experiments in which we used ConSeq together with Daikon [17] show that ConSeq can detect complicated concurrency bugs that previous tools cannot (e.g., a bug involving 11 threads and 21 shared variables). The performance of ConSeq is suitable for in-house testing.

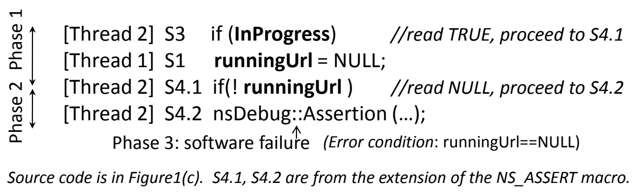
2. Background

A concurrent program's execution is a mix of inter-thread communication and intra-thread calculation. Consequently, in rough terms the error-propagation of concurrency bugs consists of three phases, as shown in Figure 3.

Phase 1 is the triggering phase. A concurrency bug is triggered by a specific execution order among a set of shared memory accesses (referred to as a *buggy interleaving*).

Phase 2 is the propagation phase. It starts with some read instructions that fetch problematic values from shared memory locations as a result of the buggy interleaving. In this paper, we will refer to these instructions as **critical reads**. The effect of these val-

[2] We skip two types of failures that have been thoroughly studied: deadlocks [24, 65] and crashes with memory errors [66].



```
[Thread 2]  S3    if (InProgress)        //read TRUE, proceed to S4.1
[Thread 1]  S1    runningUrl = NULL;
[Thread 2]  S4.1  if(! runningUrl )      //read NULL, proceed to S4.2
[Thread 2]  S4.2  nsDebug::Assertion (…);
```

Phase 1
Phase 2
Phase 3: software failure (Error condition: runningUrl==NULL)
Source code is in Figure1(c). S4.1, S4.2 are from the extension of the NS_ASSERT macro.

Figure 3. Error propagation in a concurrency bug.

ues begins to propagate, usually within one thread, through data and control dependences. Note that the corresponding thread may read many shared variables during the error-propagation phase, but only those that impact the failure through control/data dependences are considered to be 'critical'. In this paper, we measure the **distance** of error propagation through data- or control-dependence edges. During this phase, some variable values begin to deviate from the correct program state. These are called *errors*. Note that races and atomicity violations do not belong to our definition of an error.

Phase 3 is the failure phase. Propagation leads to more severe errors, and finally to an error with an externally visible symptom (i.e., a **failure**) such as a crash or a hung process.

To simplify the discussion, the rest of the paper will refer to instructions where failures/errors occur as **failure/error sites**. We refer to the same set of instructions as **potential error/failure sites** during correct runs, because they indicate sites of errors that might occur in a different interleaving. We sometimes do not differentiate failure sites from error sites, because failures can be considered to be externally visible errors. Finally, we refer to the conditions that judge whether a program has deviated from its correct states as **error conditions**, as depicted in Figure 3.

3. Error-propagation study

This section reports on a study of the error-propagation process of real-world concurrency bugs, which have not been well-studied previously. The characteristics presented here have guided our design of ConSeq.

We manually studied a set of 70 non-deadlock concurrency bugs that have been collected in previous work [32] from four widely used open-source C/C++ applications (Mozilla, MySQL database server, Apache web server, and OpenOffice), and made the following observations.

Observation 1 Concurrency bugs' failures mostly occur in one thread. This is true for 66 out of the 70 examined bugs. The only exceptions are four MySQL bugs that cause inconsistent behavior between two threads: the thread that completes a database operation first finishes its logging second. The behavior of each thread is correct. Only when comparing them with each other would we notice the misbehavior—the global log puts them in a different order from what really happened. For most bugs, propagation starts with multiple threads and ends with one thread, as shown in Figure 2.

Observation 1 implies that bug detection can be divided into separate stages of concurrency analysis and sequential analysis.

Observation 2 The failure patterns of concurrency bugs are similar to those of sequential bugs. We see five patterns—assertion-violation, error-message, incorrect outputs, infinite loop, and memory bugs—that contribute to 65 out of 70 failures. Errors become externally visible for reasons similar to ones for sequential bugs.

Observation 2 implies that it is easy to identify potential failure sites for concurrency bugs.

Observation 3 The propagation distance is usually short (referred to as the 'short-propagation heuristic'). For 59 out of the 70 bugs, after the last critical read, failure occurs before the current function exits. This trend is consistent across different applications and different types of failures. Of course, the distance is also frequently more than one data/control dependence step.

The rationale behind this observation is that there are usually operations within a few dependence steps that have the potential to cause an internal error to be visible externally. These include pointer operations, I/Os, and sanity checks inserted by programmers. Our observation about concurrency bugs is consistent with previous observations about other types of software/system defects [21, 30] that have guided previous work on failure recovery [49] and software replay [46].

Observation 3 implies that it is not difficult to look for causes based on consequences. Of course, we can also see that the approach of looking for causes from consequences is not a panacea for the challenges involved in detecting concurrency bugs. Those concurrency bugs that have long propagation distances will be challenging to detect.

4. Overview of ConSeq

As shown in Figure 4, ConSeq uses a combination of static and dynamic analyses. It uses the following modules to create an analyzer that works backwards along potential bug-propagation chains.

Error-site identifier: this static-analysis component processes a program binary and identifies instructions where certain errors might occur. For example, a call to `__assert_fail` is a potential assertion-violation failure site. Currently, ConSeq identifies potential error sites for five types of errors (Section 5). Developers can adjust the bug-detection coverage and performance of ConSeq by specifying specific types of error sites on which to focus.

Critical-read identifier: this component uses static slicing to find out which instructions that read shared memory are likely to impact a potential error site. Note that static analysis is usually not scalable for multi-threaded C/C++ programs. By leveraging the short-propagation characteristic of concurrency bugs and the staged design of ConSeq, our module is scalable to large C/C++ programs (Section 6).

Suspicious-interleaving finder: this dynamic-analysis module monitors one run of the concurrent program, which is usually a correct run, and analyzes what alternative interleavings could cause a critical read to acquire a different and dangerous value (Section 7). By leveraging the characteristics of concurrency bugs' root causes, this module is effective for large applications. Via this module, ConSeq generates a bug report, which provides a list of critical reads that can potentially read dangerous writes and lead to software failures. Critical reads, dangerous writes, and the potential failure sites are represented by their respective program counters in the bug report. Additionally, the stack contents are provided to facilitate programmers' understanding of the bug report.

Suspicious-interleaving tester: this module tries out the detected suspicious interleavings by perturbing the program's re-execution (Section 8). It helps expose concurrency bugs and thereby improves programmers' confidence in their program. Via this module, ConSeq prunes false positives from the bug report, and extends the report of each true bug with how to perturb the execution and make the bug manifest.

Note that the boundaries of ConSeq's static and dynamic analysis are not fixed. Making the bug-detection technique scalable and applicable to large C/C++ applications is a principle in ConSeq's design. ConSeq uses dynamic analysis to refine static-analysis results, and static analysis also takes feedback from run-time information.

Before diving into the technical details of ConSeq, we use the multi-variable concurrency bug shown in Figure 1(c) as an example to demonstrate the work flow of ConSeq. When we apply ConSeq to the binary of the Mozilla mail client, ConSeq's *error-site identifier* identifies 200 assertions. One of them is the instruction

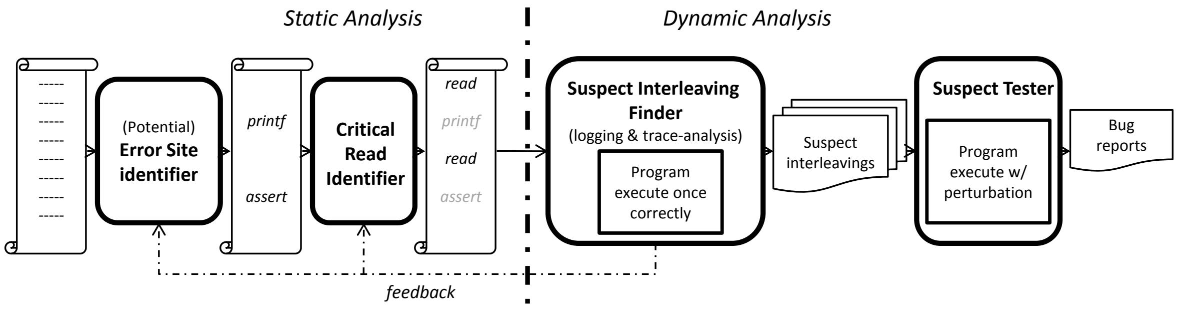

Static Analysis
Dynamic Analysis
(Potential) Error Site identifier
printf
assert
Critical Read Identifier
read
printf
read
assert
Suspect Interleaving Finder
(logging & trace-analysis)
Program execute once correctly
Suspect interleavings
Suspect Tester
Program execute w/ perturbation
Bug reports
feedback

Figure 4. An overview of the ConSeq architecture.

0x4f81d (i.e., the assertion-failure call site corresponding to S4 in Figure 1(c)). Next, ConSeq's *critical-read identifier* statically analyzes the control/data dependences leading to each assertion identified above. In particular, instruction 0x4f7f2 (i.e., the read of `runningUrl` in statement S4 in Figure 1(c)) is identified for the assertion site 0x4f81d. The application is then executed. Not surprisingly, no error occurs during the execution. ConSeq analyzes the 31 executed critical reads one by one. It identifies an alternative interleaving that might cause instruction 0x4f7f2 to read an assertion-violating value, `NULL`, defined by instruction 0x8062e5 (S1 in Figure 1(c)). Finally, ConSeq's *suspicious-interleaving tester* executes the program again and triggers a failure. In terms of users' involvement, ConSeq only requires a user to provide one thing: a test suite. Users are also allowed to provide a list of function names of interest (such as the customized error-message function). ConSeq then will automate the whole bug-finding process described above.

5. Identify potential error sites

The error-site identification module has three goals: (i) to identify potential error sites automatically, (ii) to identify them before an error occurs, and (iii) to accomplish (i) and (ii) with good accuracy and coverage. This module provides the starting points for ConSeq's backward concurrency-bug detection strategy and directly affects the false-positive and false-negative rates of ConSeq. To achieve its goals, ConSeq follows two design principles:

(1) Use static analysis instead of dynamic analysis. Errors rarely occur during monitored runs of concurrent programs. Static analysis can go beyond what occurs during a single execution.

(2) Exploit the failure patterns of software bugs. Concurrency bugs, fortunately, have similar failure patterns as sequential bugs, which are well-studied and well-understood.

5.1 Identifying explicit failure sites

Failures of non-deadlock concurrency bugs can be covered by five patterns that they share with sequential bugs (Section 3). ConSeq identifies each pattern as follows.

Infinite Loop: For non-deadlock bugs, infinite loops in one thread are the main causes of hangs (an example is shown in Figure 1(b)). Every back-edge in a loop is a potential site for this type of failure. ConSeq identifies strongly connected components (SCCs) that are potential failure sites for infinite-loop hangs by checking whether any shared-memory read is included in the backward slice of each back-edge in an SCC. To identify nested loops, CodeSurfer/x86 implements *Bourdoncle*'s algorithm [6], which recursively decomposes an SCC into sub-SCCs, etc.

Assertion Violations: Assertion violations (Figure 1 (c)) are a major source of program crashes. Fortunately, it is a common practice of developers to place assertions in their code. Moreover, assertions are able to specify certain other types of errors. In C/C++ programs, a call to gcc's `assert` library function is translated to an if statement whose else-branch contains a call to `__assert_fail`. The call sites on `__assert_fail` are considered to be potential failure sites. Some applications use customized assertions, such as `nsDebug::Assertion` in Mozilla. ConSeq also considers those call sites to be potential failure sites.

Memory Errors: Invalid memory accesses are another major source of program crashes, especially in C/C++ programs (e.g., the NULL-pointer dereference shown in Figure 1(a)). ConSeq incorporates a module to identify every pointer dereference as a potential memory-error site. Because this specific type of error has been thoroughly studied in recent work [66], we focus on other types of failures and corresponding concurrency bugs in this paper.

Incorrect Outputs: Most non-fail-stop software failures occur when the software generates incorrect outputs or totally misses an output. ConSeq considers a call to an output function, such as `printf` and `fprintf`, as a potential incorrect-output failure site. Some applications have special output functions, such as MySQL's `BinLog::Write`. ConSeq allows developers to specify application-specific output functions in a text file. ConSeq reads the text file and identifies call sites on the specified functions.

Error Messages (consistency-check sites): Consistency checks have an interesting role in concurrency bugs. They are usually **not** designed for catching synchronization bugs, and simply reflect a developer's wish to enforce some important correctness property. Luckily, however, for many complicated concurrency bugs, there are warning signs long before the ultimate failure arises. As a result, the error-propagation distance is greatly shortened and backward bug-detection becomes much easier due to error messages. *Writing such consistency checks has been a common practice of programmers for a long time [28], and the presence of consistency checks can greatly help the approach taken by ConSeq*.

ConSeq identifies calls to functions that print error messages as potential failure sites. These include both library functions, such as `fprintf(stderr,...)`, and application-specific routines, such as the `NS_WARNING` in Mozilla and `tr_err` in Transmission (a BitTorrent client). ConSeq allows developers to specify these error-reporting functions in a file. ConSeq reads this file and identifies call sites on all these functions. In our experience, most applications only have a few (usually just one or two) error-reporting routines. Therefore, we believe it will not be a big burden for developers to write down these functions.

In the case of assertion failures and error messages, a condition that indicates whether the value acquired at a given site is correct or not is obtained as a by-product. This condition is used to improve the accuracy of ConSeq's bug-detection capabilities (Section 7).

5.2 Inferring implicit error sites

As discussed above, consistency checks added by developers are very helpful in ConSeq's method for bug detection. What if developers did not provide any consistency checks?

Interestingly, a lot of research on sequential programs has faced this problem before, and some solutions have been proposed. For instance, Daikon [17] is a tool that infers likely program invariants based on evidence provided by (correct) training runs. Daikon's most advanced features allow for inference among derived variables, as well as set relations and arithmetic relations between array elements. In this respect, Daikon can automatically provide information that is similar to the consistency checks manually added by developers. We can treat those places where Daikon identifies program invariants to be potential error sites.

Specifically, we first apply Daikon to the target software. Daikon's frontend logs run-time variable values at program points selected by Daikon. Daikon's backend processes the log and outputs a list of {program-point, invariant} pairs.

ConSeq checks every global read instruction *I* that reads global variable *v*. If Daikon has identified an invariant involving *v* right before *I*, ConSeq identifies *I* as a potential invariant-violation site.

One implementation challenge we encountered is that the default frontend, kvasir-dtrace, of Daikon's academic version only collects information at function entries and exits. As a result, we cannot obtain invariants at the granularity of individual instructions. With the help of the Daikon developers, we tried two ways to get around this problem. For small applications in our experiments, we manually inserted dummy functions before every global-variable read. For large applications in our experiments, we replaced Daikon's default front-end with our own PIN tool. This PIN tool collects run-time information before every global-variable read, and outputs this information in the input format used by kvasir-dtrace. By this means, the Daikon backend can process the Pin tool's output and generate invariants.

For large applications, one potential concern is that Daikon could identify a huge number of invariants, which could impose a large burden on ConSeq's critical-read identifier, suspicious-interleaving finder, and suspicious-interleaving tester. Fortunately, Daikon provides ranking schemes [18] to identify important invariants. ConSeq leverages the ranking mechanism to focus on the most important invariants.

In summary, ConSeq currently focuses on five types of potential failure/error sites. Except for the potential error sites inferred by Daikon, all sites are identified by statically analyzing the program.

6. Identifying critical reads

The goal of the critical-read identification module is to identify *critical-read* instructions that are likely to impact potential error sites through data/control dependences. It uses static slicing to approximate (in reverse) the second propagation phase of a concurrency bug, as shown in Figure 2. There are two major design principles for this module:

1. Use *static* analysis rather than dynamic analysis to identify which instructions *may* affect an error site. ConSeq is different from failure-diagnosis tools. It aims to expose concurrency bugs **without any knowledge of how they may arise or even if they exist**, so its analysis cannot be limited to any specific monitored run. Specifically, ConSeq uses *static slicing* for this purpose.
2. Only report instructions with short propagation distances as critical reads. Computing the complete program slice, e.g., all the way back to an input, is complicated and also unnecessary for ConSeq. ConSeq leverages the short-propagation characteristic of concurrency bugs (Section 3) to improve bug-detection efficiency and accuracy.

6.1 General issues

We had to make several design decisions that are general to all types of error sites:

Must critical-read instructions access shared memory? Instructions that read thread-local variables could be of interest for sequential bug detection, but not for concurrency bug detection, because their values cannot be directly influenced by interleavings. To get rid of these instructions, ConSeq first uses static analysis to filter out as many stack accesses as possible. ConSeq's run-time monitoring will proceed to prune out the rest of the stack accesses. Of course, it is possible for threads to share values using the stack, although it is rare in practice. Escape analysis would be able to identify these special stack accesses, and make ConSeq more accurate. We leave this as future work.

Shall we consider inter-thread control/data dependences? Multi-thread static slicing is much more difficult than single-thread slicing. Fortunately, because ConSeq's design separates the propagation steps in a concurrency bug into inter-thread and intra-thread phases, here only single-thread dependence analysis is needed to identify critical reads. All analyses involving multi-thread interleavings will be conducted in the suspicious-interleaving finder (Section 7).

How to set the propagation-distance threshold? In accordance with the short-propagation heuristic, ConSeq only reports read instructions whose return values can affect the error sites through a short sequence of data/control dependences. Our static-slicing tool provides the slice, together with the value of the shortest distance to the starting point of the slice, for each instruction of the slice. An example is shown in Figure 5. ConSeq provides a tunable threshold *MaxDistance* for users to control the balance between false negatives and false positives. By default, ConSeq uses 4 as *MaxDistance*. A detailed evaluation is presented in Section 10. We will explore other metrics for propagation distance in the future.

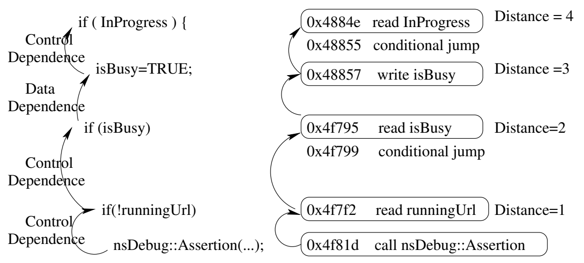


Figure 5. Static slicing of machine code (right) and the distance calculation.

How to reuse the analysis results across inputs? Because ConSeq uses static instead of dynamic analyses, the results from this module, as well as those from the error-site identifier, can be reused for different inputs. Our current static slicer analyzes one object file at a time. To speed up the analysis when there are only a few inputs, we first process those object files that these inputs would touch.

Customization for different types of error sites

ConSeq customizes the analysis for each type of error site:

- Each consistency-check error site is a 'call' instruction that calls a standard or custom error-reporting routine. ConSeq directly applies slicing for that instruction. For a 'call' instruction, the first step backward is always through a control dependence, followed by a sequence of control and data dependences.
- Each assertion-failure site is a 'call' to the `__assert_fail` library routine. We handle it in the same way as a consistency-check error site.
- Each invariant-violation failure site is an instruction that reads heap or global variables. No customization is needed. ConSeq directly applies control-and-data slicing for each of these instructions.
- Each incorrect-output site is a '`call`' instruction to output functions. Before applying static slicing, we first use a simple

static analysis to identify all instructions that push argument values onto the stack (sometimes via `push` and sometimes via `mov`).[3] We add those instructions into the slice (at distance 1) and apply slicing to these instructions and the original `call`
- Each infinite-loop site involves a jump instruction that conditionally jumps out of a loop. Among instructions that are on the slice, we only keep those that are repeatedly executed inside the loop body, because only those instructions could lead to the loop executing repeatedly.

6.2 Static slicing details

Program slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions may consist of elements that are not textually contiguous [59]. A *backward slice* of a program with respect to a set of program elements S consists of all program elements that might affect (either directly or transitively) the values of the variables used at members of S. Slicing is typically carried out using *program dependence graphs* [22].

CodeSurfer/x86. ConSeq uses backward slicing to identify shared memory reads that might impact each potential error site. To obtain the backward slice for each potential error site, it uses CodeSurfer/x86 [2], which is a static-analysis framework for analyzing the properties of x86 executables. Various analysis techniques are incorporated in CodeSurfer/x86, including ones to recover a *sound approximation* to an executable's variables and dynamically allocated memory objects [3]. CodeSurfer/x86 tracks the flow of values through these objects, which allows it to provide information about control/data dependences transmitted via memory loads and stores.

Side-Stepping Scalability Problems. To avoid the possible scalability problems that can occur with CodeSurfer/x86 due to the size of the applications used in evaluating ConSeq, we set the starting point of each analysis in CodeSurfer/x86 to the entry point of the function to which a given potential error site belongs, instead of the main entry point of the program. By doing so, CodeSurfer/x86 only needs to analyze the functions of interest and their transitive calls rather than the whole executable. Thus the static analyses time grows roughly linearly in the number of functions that contain error sites. This makes ConSeq much more scalable, as will be illustrated in Section 10.

This approach is applicable in ConSeq because—based on the observation that the error-propagation distance is usually short, as discussed in Section 3—ConSeq only requires a *short* backward slice that can be covered in one procedure. The backward-slicing and other analysis operations in CodeSurfer/x86 are, however, still context-sensitive and *interprocedural* [22]. Moreover, to obtain better precision from slices, each of the analyses used by CodeSurfer/x86 is also performed interprocedurally: calls to a subprocedure are analyzed with the (abstract) arguments that arise at the call-site; calls are not treated as setting all the program elements to $\top$.

Analysis Accuracy. To obtain static-analysis results that over-approximate what can occur in any execution run, all the program elements (memory, registers, and flags) in the initial state with which each analysis starts are initialized to $\top$, which represents any value. Such an approximation makes sure that no critical read will be missed by ConSeq at run time. Of course, some instructions could be mistakenly included in the backward slice and be wrongly treated as critical reads. Fortunately, our short-propagation-distance heuristic minimizes the negative impact of over-approximation. In practice, we seldom observe the inaccuracy caused by this over-approximation.

Finally, the CodeSurfer/x86 framework has information about every direct calls' call-sites. Therefore, if needed, it can also support backward slicing that starts at the entry of a procedure and backs up into the callers.

[3] Figuring out the parameters to a call in the binary is easy for the CDECL calling convention. If we need to process GCC __fastcalls, we will need to go back to analyzing the source code. The current implementation of ConSeq does not handle __fastcalls.

7. Identifying suspicious interleavings

The module for finding suspicious interleavings focuses on the first phase of concurrency-bug propagation. ConSeq monitors a program's (correct) execution, collects a trace using binary instrumentation, and analyzes the trace to decide whether a different interleaving could change the dynamic control/data dependence graph and generate a potentially incorrect value at a critical read.

Because it is impractical to check all potential interleavings and all potential dynamic control/data dependence graphs [23], ConSeq leverages the short-propagation characteristic (Section 3) and the widely used shallow-depth heuristic (i.e., the manifestation of most concurrency bugs involves only two or three shared-memory accesses) [7, 38, 39, 44]. It examines writes that are *one data-dependence step backward* from each critical read r, and looks for suspicious interleavings that could make r obtain a potentially incorrect value written by a write access that is different from the one that occurred in the monitored run. The algorithm in ConSeq is neither sound nor complete. Rather, ConSeq tries to balance generality, simplicity, and accuracy.

7.1 The core analysis

We formalize the key question this portion of ConSeq has to answer as follows: in a concurrent program's execution trace T, a read instruction r gets a value defined by a write access w; we ask whether r can read a value defined by a different write w' in an alternative interleaving.

To realize a w'–r data-dependence, three conditions have to be satisfied. First, w' and r need to access the same memory location m. This condition is fairly easy to check, as long as ConSeq records the addresses of the locations touched by memory accesses.

Second, w' needs to execute before r. This condition can be prohibited by barrier-style synchronizations. Therefore, ConSeq monitors `pthread_create/join`, `pipe`, and `barrier` at run-time to maintain vector-clock time-stamps for each thread and hence each access. A w'–r dependence is infeasible if r has a smaller time-stamp than w'. ConSeq computes the vector-clock time-stamps in a similar way as traditional happens-before race detectors [42]. ConSeq does not update time-stamps according to `lock/unlock` operations, because these operations do not provide any execution-order guarantees.

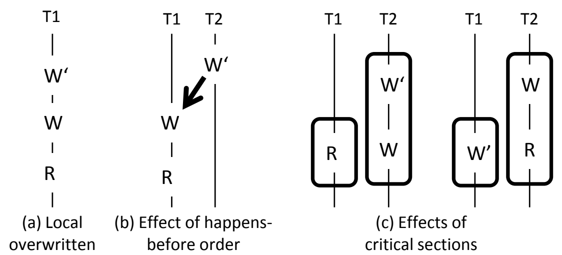

T1
W'
W
R
(a) Local overwritten
T1
T2
W'
W
R
(b) Effect of happens-before order
T1
T2
R
W'
W
(c) Effects of critical sections
T1
T2
W'
W
R

Figure 6. A value written by W' may never reach R

Third, the value written by w' to m is not overwritten before it reaches r. There are three situations in which an overwrite always happens, as demonstrated in Figure 6. The first is due to intra-thread program logic, when there is another write w to m between w' and r in the same thread as shown in Figure 6(a). The second is due to barrier-style synchronization, as shown in Figure 6(b). That is, synchronization operations, such as `barrier` or `pthread_create/join`, force w' to always execute before another write w and w to always execute before r. The third is due to mutual exclusion, as shown in Figure 6(c). When w' is followed by w in a critical section from which r excluded, the result of w' can never reach r. The situation is similar when r is preceded by w in a critical section from which w' is excluded. As long as the trace includes sufficient information about lock operations, ConSeq can analyze all of these situations.

Pseudo-code for the method described above is given as Algorithm 1.

Algorithm 1 ConSeq identify suspicious interleavings

```
Require: write access $w$
Require: write access $w'$
Require: read access $r$
Require: $w$,$w'$,$r$ access the same shared memory address
Ensure: return true if r can read a value from $w'$, false if not
 1: /*
    Time-stamp comparison is based on the happens-before relationship and vector-clock time-stamps
    */
 2: if $r$.time-stamp $<$ $w'$.time-stamp then
 3:     /*$r$ happens before $w'$ */
 4:     return false
 5: end if
 6: if $w'$.time-stamp $<$ $w$.time-stamp $<$ $r$.time-stamp then
 7:     /*$w'$ is overwritten by $w$ */
 8:     return false
 9: end if
10: if $w$ is executed before $r$ in a critical section $CS1$, $w'$ is in critical section $CS2$, $CS1$ and $CS2$ are from different threads, and $CS1$ is mutually exclusive from $CS2$ then
11:     return false
12: end if
13: if $w'$ is executed before $w$ in a critical section $CS1$, $r$ is in critical section $CS2$, $CS1$ and $CS2$ are from different threads, and $CS1$ is mutually exclusive from $CS2$ then
14:     /*$w'$ is overwritten by $w$ */
15:     return false
16: end if
17: /*Report feasible in all the other cases */
18: return true
```

7.2 The complete algorithm and extensions

ConSeq uses binary instrumentation to monitor three types of operations at run-time: critical-read instructions, instructions that write global and heap variables, and synchronization operations. For each memory-access instruction, ConSeq records the program counter, the address of the accessed memory, and the value of the accessed memory location before the read or after the write. For each lock operation (`pthread_mutex_(un)lock`), ConSeq records the address of the lock variable. For each barrier-style synchronization (`pthread_create/join`, `pipe`, `barrier`,etc.), ConSeq updates the vector time-stamps of every thread. ConSeq uses one trace file for each thread to avoid slow global synchronization. Given these pieces of information, ConSeq can easily analyze the trace and find out all feasible w'–r dependences.

ConSeq also extends the basic algorithm in three ways.

First, ConSeq records the values read by r and written by w' during the correct run, denoted by v' and v, respectively. If the two values are the same, ConSeq does not report a suspicious interleaving. To further prune false positives, ConSeq also evaluates v' against the assertion/error-condition before reporting a suspicious interleaving, using a symbolic-execution module inside ConSeq.

Second, the basic algorithm cannot be directly applied for detecting infinite loops. Suppose that r is a critical read that is associated with a potential infinite-loop site. During the monitored run, ConSeq records the write w and its value v that are read by the last dynamic instance of r right before the loop terminates. Now suppose that the basic algorithm identifies an alternative interleaving in which this specific instance of r can receive a different value from an alternative write w'. This condition is insufficient to conclude that this interleaving is suspicious. If w is executed after w', another instance of r in a later iteration of the loop can still receive v from w and terminate the loop. Therefore, for each alternative write w' identified by the basic algorithm, ConSeq further compares the happens-before time-stamps between w and w'. An infinite-loop suspect is reported when w' is strictly ordered after w and when w' is concurrent with w.

Third, interleavings could make a critical read r execute too early and receive an uninitialized value. ConSeq also reports these cases as suspicious interleavings.

Discussion. There are several sources of inaccuracy in our analysis that can cause false positives and negatives. One is that the value written by a write w' might vary in different runs. Another is that interleavings could change the control flow and cause inaccuracy of our analysis. Finally, ad-hoc synchronization has been a problem for almost all predictive concurrency-bug-detection tools. We leverage our static analysis component, which identifies loops, back-edge jumps, and backward slices of back-edge jumps, to identify one type of common ad-hoc synchronization (one thread spins on a while-flag to wait for another thread). The identification algorithm is similar to previous work [61]. After identifying this type of ad-hoc synchronization, ConSeq treats occurrences as traditional barrier-style synchronizations.

8. Exercising suspicious interleavings

The input to ConSeq-tester, the module for testing suspicious interleavings, is a list of data dependences, represented as write/read pairs (w_{bad}–r). The goal is to exercise suspicious interleavings that can realize these suspicious data dependences, so that we can either reliably trigger the bugs or prune them as false positives.

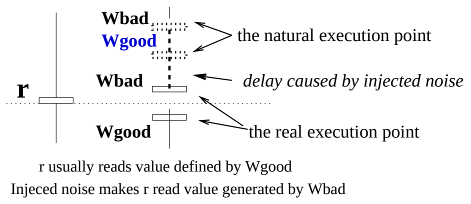

Wbad
Wgood
the natural execution point
Wbad
delay caused by injected noise
r
Wgood
the real execution point
r usually reads value defined by Wgood
Injected noise makes r read value generated by Wbad

Figure 7. Exercising a suspicious interleaving.

To achieve this goal, ConSeq uses a testing technique that has been used in several previous bug-detection tools [44, 53]. Specifically, ConSeq instruments the program binary and inserts conditional delays with time-outs before every r and w_{bad} instructions. ConSeq then re-executes the program with the original input. Because ConSeq is used during in-house testing, the input is available. At run time, the instrumented code either suspends for a while the thread that is going to execute w_{bad}, to wait for the arrival of r in another thread, or suspends for a while the thread that is going to execute r, to wait for the arrival of w_{bad}. When both w_{bad} and r are ready to execute, the instrumented code will force the program to execute w_{bad} immediately followed by r. Therefore, the probability that those w_{bad}–r dependences occur is significantly improved. An example of how ConSeq-tester exercises a suspicious interleaving is shown in Figure 7.

We have encountered two interesting issues in ConSeq.

First, w_{bad} and r might be from the same thread. The basic scheme shown in Figure 7 does not work for this case, because ConSeq will not see r coming when it blocks a w_{bad} operation that is from the same thread as r. ConSeq's suspicious-interleaving identification module marks these cases during trace analysis. During testing, instead of blocking w_{bad}, ConSeq will let it proceed and block any following writes that touch the same memory location that w_{bad} accesses, until r is executed. Second, sometimes w_{bad} and r are protected by the same lock. In those cases, ConSeq inserts a delay before the thread enters the corresponding critical section.

Like many previous concurrency-bug validation tools [40, 44, 66], ConSeq can significantly increase the probability that a concurrency bug manifests, but it cannot provide a 100% guarantee to provoke every bug. In Section 10, however, we will see that ConSeq performs well in practice.

9. Experimental Methodology

Bug-ID	Symptoms	Application	LOC
Aget1*	Wrong output	Aget-0.4.1	1.1K
FFT	Wrong output	FFT	1.2K
MySQL1	Miss log	MySQL-4.0.12	681K
Moz1	Assertion	Mozilla-1.7	1.2M
MySQL2	Assertion	MySQL-4.0.16	654K
Trans	Assertion	Transmission	95K
Moz2	Error message	Mozilla JS-1.5	87K
Moz3	Error message	Mozilla	N/A
MySQL3	Error message	MySQL-5.0.16	1.6M
MySQL4	infinite-loop	MySQL-5.0.41	1.6M
OO	infinite-loop	OpenOffice	N/A
Cherokee-0.99.48*, web server			96K
Click-1.8.0*, modular router			290K

Table 2. Applications and Bugs (Mozilla-JS is the Mozilla Javascript Engine; Cherokee-0.99.48 and Click-1.8.0 are both the latest versions and previously had no known buggy inputs; Moz3 and OO are extracted from old versions of Mozilla and OpenOffice that can no longer compile. ***:ConSeq detected new bugs in Aget, Cherokee, and Click.**)

ConSeq's dynamic modules are implemented using the PIN [37] binary-instrumentation framework. The experiments are carried out on an 8-core Intel Xeon machine running Linux version 2.6.18.

We evaluated ConSeq on 8 widely used C/C++ applications. This includes two server applications (the MySQL database and the Cherokee web server), two client applications (Transmission BitTorrent client and Mozilla), two desktop applications (Aget file downloader and OpenOffice), one router (Click [13]), and one scientific application kernel (FFT [60]).

Input design is usually out of the scope of dynamic bug detection [33, 51] and interleaving testing [39, 44, 53], and ConSeq is no different. The intended usage scenario is that ConSeq will be applied to a test suite during in-house testing to expose hidden interleaving errors from (apparently) non-buggy runs on inputs provided by developers or testers. Our experiments were designed to provide insight on the following two questions:

(1) Can ConSeq handle a wide range of types of concurrency bugs? To address this question, and to evaluate ConSeq's bug-detection capability in comparison with traditional bug-detection tools, we used a large set of concurrency bugs that cover different failure symptoms from different applications. In particular, we took 11 concurrency bugs—which cover assertion failures, hangs, wrong outputs, and error-message problems—from the change logs and bug databases of 6 applications (the first 11 lines of Table 2). In these experiments, to drive ConSeq's bug-detection process we used inputs that were known to have the *potential* of triggering the bug. Our experiments did not leverage any information about the bugs, other than the known inputs. In fact, **none** of the bugs *ever* manifested during the runs that ConSeq performs to generate execution traces for subsequent bug-detection analysis. This methodology is consistent with that used in many previous studies [44, 53]. We will see that ConSeq was able to handle a wide range of types of concurrency bugs (detecting 10 of the 11 bugs).

(2) Can ConSeq find new bugs in the setting of in-house testing (i.e., bugs are not previously known, and inputs are supplied by knowledgeable users)? To mimic the setting of in-house testing, we applied ConSeq to the latest versions of the Cherokee web server [10] and the Click [13] modular router, using test inputs provided by their developers. We were not aware of any concurrency bugs in these two programs. We will see that ConSeq found concurrency bugs in them. Note that these experiments were not started until ConSeq's design and implementation were completely finished. The ability of ConSeq to detect such unknown concurrency bugs also demonstrates the effectiveness of heuristics like the short-propagation heuristic.

Our evaluation of false positives and performance overhead completely executes each input (or set of client requests) from the beginning to the end. The reported performance numbers are the average across 5 runs. The reported false-positive numbers are stable across the multiple runs that we tried. By default, we set *MaxDistance* to 4. We also evaluate false-positive and false-negative results under different *MaxDistance* settings. ConSeq-Daikon demands special setup, and is discussed separately in Section 10.5.

For comparison, we also evaluated two state-of-the-art cause-oriented approaches to detecting concurrency bugs under the same setting. *Race* is a lock-set–happens-before hybrid race detector, commonly known as Helgrind, implemented as part of the open-source bug-detection tool Valgrind [41]. *Atom* [44] detects the most common type of atomicity bug (two accesses in one thread unserializably interleaved by another thread [32, 33, 56]). Similar to ConSeq, these two detectors aim to detect bugs from correct runs.

10. Experimental Results

10.1 Overall bug-detection results

Table 3 shows the overall bug-detection results. As we can see, ConSeq has good coverage in bug detection. It detected 10 out of the 11 bugs. *Race* and *Atom* only correctly detected 3 and 4 bugs, respectively.

Aside from the bug in Aget listed in the Table 2, ConSeq detected two new bugs in Aget that have never been reported before (one by tracing back from a `printf` call site and one by finding a violation of a (candidate) invariant identified by ConSeq-Daikon). In MySQL-5.0.16, ConSeq detected an infinite-loop concurrency bug initially reported in MySQL-5.0.41, which shows that the bug actually existed in the older version and can be triggered using a different input. Our analysis of Cherokee-0.99.48 and Click-1.8.0 used the basic inputs provided in the applications' test suites, and ConSeq discovered bugs in them as we will see in Section 10.3.

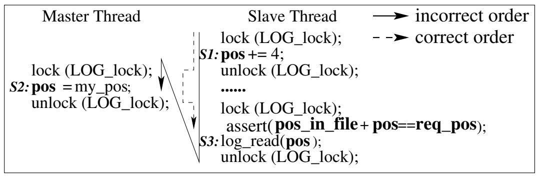
Atom targets single-variable atomicity violations that involve three accesses, and cannot detect concurrency bugs caused by other interleaving patterns, such as the bugs in Moz-1 (a multi-variable atomicity violation), MySQL-2 and OO (anti-atomicity violations where the software behaves correctly only when a certain code region in a thread is not atomic), Moz-3 (an atomicity violation

Bug-ID	ConSeq Detected	*Race* Detected	*Atom* Detected
Aget1			
FFT	✓		✓
MySQL1	✓		✓
Moz1	✓		
MySQL2	✓		
Trans	✓		
Moz2	✓	✓	✓
Moz3	✓	✓	
MySQL3	✓	✓	✓
MySQL4	✓		
OO	✓		

Table 3. Bug detection results (✓: detected; Blank: not).

involving more than three accesses), MySQL-4 and Trans (order violation), etc. *Race* suffers from a similar source of false negatives as *Atom*: the root cause of many of these bugs has nothing to do with locks, and many buggy code fragments did use locks correctly (e.g., OO, MySQL-4). In addition, *Race* uses some heuristics to lower the false-positive rate (e.g., not reporting a race when earlier races are already reported on that variable), which leads to some false negatives.

ConSeq's consequence-oriented approach means that its bug-detection capabilities are not limited to any specific interleaving pattern, and thus ConSeq can detect bugs that *Race* and *Atom* cannot. Section 4 has already discussed how ConSeq detects Moz-1, the **multi-variable bug** illustrated in Figure 1(c). Figure 8 shows an **anti-atomicity** example (MySQL2). *S*3 from the slave thread wants to use the value of `pos` defined by the master thread (*S*2) to read the log. Unfortunately, *S*3 could non-deterministically execute before *S*2 and mistakenly read a value defined by its own thread, leading to the MySQL failure. The bug is obviously not a race, because all accesses are well-protected. The bug is also not an atomicity-violation bug, because MySQL executes *correctly* when the atomicity between *S*1 and *S*3 is *violated*! Furthermore, it is not a simple order-violation bug, because there are many dynamic instances of *S*1, *S*2, and *S*3. No order between *S*2–*S*1 or *S*2–*S*3 can guarantee failure. With a cause-oriented approach [34, 55], more sophisticated interleaving patterns and a large number of training runs are needed to detect this bug. In contrast, with ConSeq's consequence-oriented approach, this bug presents no special challenges. MySQL developers already put a sanity check before each log read: `assert(pos_in_file + pos == req_pos)`. Analyzing backwards from that check, ConSeq easily discovers the bug.



```
                              lock (LOG_lock);
                          S1: pos += 4;
    lock (LOG_lock);          unlock (LOG_lock);
S2: pos = my_pos;             ......
    unlock (LOG_lock);        lock (LOG_lock);
                               assert(pos_in_file + pos==req_pos);
                          S3: log_read(pos);
                              unlock (LOG_lock);
```

Figure 8. An example showing that ConSeq can detect a non-race, non-atomicity-violation bug (simplified for purposes of illustration).

New bugs detected by ConSeq: Apart from detecting the bugs described above, ConSeq also detected two more concurrency bugs that we were unaware of in Aget, and a known infinite-loop bug in a different version of MySQL than originally reported.

More interestingly, ConSeq found an output non-determinism in Cherokee and two bugs in Click. For example, one bug in Click can cause locks to be destroyed when they are still in use. ConSeq can detect this bug using any input provided in Click's test suite. Specifically, after a correct run of Click, ConSeq reported that an error message `"Spinlock:: Spinlock(): assertion '_depth == 0' failed"` could be triggered under a different interleaving. The report accurately points us to the bug. In terms of the root cause, this bug is a non-datarace order-violation bug. Neither *Race* nor *Atom* is able to detect this bug.

False negatives: There is one bug in Aget that evaded detection by all three tools. The bug is an atomicity violation that involves 11 threads and 21 shared variables. ConSeq failed to detect it because the bug involves a long propagation distance. However, with the support of Daikon, ConSeq can successfully detect it, see (Section 10.5).

ConSeq and traditional tools look at concurrency bugs from different perspectives and can miss bugs in different ways. *Race* and *Atom* have false negatives in the examples discussed above because they cannot cover certain interleaving patterns. ConSeq will inevitably miss some bugs due to missing certain types of failure sites or due to error-propagation distances that exceed the threshold used in the short-propagation heuristic. Of course, the coverage of ConSeq could be further improved in the future by adding more failure templates or tuning the *MaxDistance* threshold. It could also be helped by the use of additional invariant-inference techniques and by developers who are comfortable with adding consistency checks. In summary, ConSeq can well complement existing bug-detection approaches.

10.2 False positives

Before suspicious-interleaving testing. False positives have always been a problem in concurrency bug detection, especially for predictive bug detectors that need to analyze a huge number of potential interleavings, such as ConSeq, *Race*, and *Atom*.

Bug-ID	ConSeq				*Race*	*Atom*	Base
	OUT	ASS	ERR	LOOP			
Aget1	0	0	0	0	2	4	0
FFT	0	0	0	0	8	16	20
MySQL1	0	0	0	0	127	51	77
Moz1	0	0	0	0	26	n/a	OM
MySQL2	0	4	2	5	163	402	OM
Trans	0	2	0	0	42	33	136
Moz2	1	4	0	0	20	279	244
Moz3	0	0	0	0	0	0	0
MySQL3	0	4	3	7	714	1026	OM
MySQL4	0	1	3	0	180	552	197
OO	0	0	0	0	0	0	0
Cherokee	0	0	1	3	40	296	OM
Click	0	1	0	0	13	20	37

Table 4. False positives in bug detection. (OM means analysis runs out of memory before finish.)

As shown in Table 4, ConSeq has much better accuracy than *Race* and *Atom*,[4] exhibiting only about one-tenth the false-positive rate of the latter two. For 11 out of the 13 cases, ConSeq only has 0–5 false positives. Compared to traditional predictive bug-detection tools, ConSeq can save a lot of testing resources and manual effort by developers.

The main reason that ConSeq reports fewer false positives than traditional approaches is that its consequence-oriented approach has made it much more focused. To validate this, we also measured the false-positive rate for ConSeq with identification of critical reads turned off. The numbers are roughly comparable to the ones

[4] We conducted manual validation for randomly sampled *Race* and *Atom* bug reports.

for *Atom*, as shown by the last column ('Base') in Table 4. Actually, if not guided by potential failure sites and critical reads, the analysis runs out of memory before finishing for several MySQL and Mozilla workloads, because the interleaving space is huge. This is exactly why ConSeq identifies critical read instructions based on potential failure patterns.

The false positives of ConSeq are of two types: (1) Unidentified customized synchronization operations make a suspicious interleaving infeasible. This reason is responsible for all but 3 cases. (2) A different but still correct value is read at a critical read *r*. This is responsible for 3 false positives.

After suspicious-interleaving testing. ConSeq-tester prunes out all the false positives discussed above with one false negative. Specifically, ConSeq-tester successfully makes the 9 bugs detected by ConSeq in Table 3 manifest. Unfortunately, of the two new bugs detected by ConSeq in Click, one cannot be automatically exposed by ConSeq-tester. Its manifestation requires a complicated sequence of branches to be followed, involving multiple branch points in Click.

For those bugs that can be automatically exposed by ConSeq-tester, their manifestation can all be reliably repeated by inserting delays at the same places recorded by ConSeq-tester, which can help programmers perform further diagnoses.

10.3 Detailed bug-detection results

Sensitivity of MaxDistance: In ConSeq, the *MaxDistance* threshold affects how many read instructions are considered 'critical'. We measured the false positives and false negatives of ConSeq under different *MaxDistance* settings. The total number of false positives gradually increases, as does the bug-detection capability. Adding all 13 bug-detection runs together, ConSeq reports 25, 33, 37, 41, 61 false positives in total, with *MaxDistance* set to 1, 2, 3, 4, and 5, respectively. ConSeq detects 6, 8, 9, 10, and 10 out of the tested 11 bugs, with *MaxDistance* set to 1, 2, 3, 4, and 5, respectively. These numbers demonstrate the usefulness of static slicing in ConSeq: if ConSeq only looks at shared variables right at the failure site, almost half of the bugs will be missed.

	Object File Size	# of Error Sites	# of Critical Reads
Aget1	56K	86	49
FFT	33K	52	111
MySQL1	15M	2137	8562
Moz1	4.9M	142	397
MySQL2	14M	487	1369
Trans	1.2M	158	232
Moz2	1.4M	856	929
Moz3	5.7K	1	1
MySQL3	25M	1349	2020
MySQL4	27M	867	2341
OO	13K	26	75
Cherokee	2.6M	424	1261
Click	24M	386	1365

Table 5. Potential error sites and critical reads.

Potential error sites and critical reads. Table 5 shows the size of the object files processed by our static-slicing tool and the number of potential failure sites and critical reads identified. As we can see, our static-analysis component can handle large applications whose object files are tens of mega-bytes.

10.4 Performance results

ConSeq is designed for in-house testing and goes through three phases. The first phase of ConSeq uses static analysis to identify potential failure sites and critical reads. This step is *not* performance-critical, because it is conducted only *once* for each piece of code. It can be re-used across different testing runs and different inputs. Even after a code modification, ConSeq only needs to re-analyze those object files that have been changed. In our experiments, the static analysis is scalable. It can finish within a couple of hours for most applications. Processing MySQL3 takes the longest time — 18392 seconds or about 5 hours.

The second phase of ConSeq takes a test input, runs it once to collect a trace, and analyzes the trace to report suspicious interleavings. At the end of this step, concurrency bugs are reported as shown in Table 3 and 4. This step would be repeated many times during in-house testing and is the most important for ConSeq's performance. Table 6 shows the results for this phase. ConSeq's runtime overheads for the four types of failure patterns are similar, so for each application in the table we only present the worst-case performance and largest trace size among these four cases. Adding the run-time and the time for off-line analysis of the trace together, ConSeq introduces execution overhead of 1.26X — 38.5X for each test input, which is suitable for in-house use. ConSeq's trace sizes are reasonably small. The biggest trace size is about 115MB in our experiments, as shown in Table 6. The peak memory consumption of ConSeq at run-time is less than 100 MB for all applications.

At the end, ConSeq also has an optional step of suspicious-interleaving testing. This step imposes less than 10–55% execution overhead for validating each bug report. Given the small false-positive rate of ConSeq, this step does not take a long time and can be omitted by developers. In our experiments, MySQL3 had the largest accumulated overhead at this step, because it has the largest number of false positives. The baseline run of MySQL3 finishes in 0.46 seconds. In total, ConSeq took 10.6 seconds for the validation step for all 15 reported suspicious interleavings and pruned out 14 false positives.

	Base Line	Run-Time Overhead (%)	Trace Analysis	Trace Size
Aget1	12.45s	26%	0.01s	24.7K
FFT	0.05s	2724%	0.01s	1.2M
MySQL1	0.18s	21.1%	0.27s	2.9M
Moz1	0.18s	444%	1.57s	36M
MySQL2	0.13s	157%	0.27s	18M
Trans	1.17s	210%	0.01s	132K
Moz2	12.0s	1065%	0.01s	490K
MySQL3	0.46s	130.2%	6.77s	67M
MySQL4	0.10s	135.5%	0.13s	17M
Cherokee	11.26s	21.28%	11.45s	115M
Click	0.02s	3846%	0.01s	80K

Table 6. Performance of trace collection and analysis (Base Line is the time for the original test run w/o any instrumentation.)

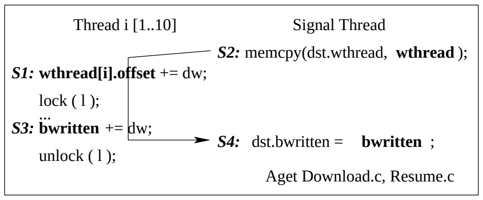
10.5 Experience with ConSeq-Daikon

As discussed in Section 5, we played some tricks to get around the granularity limitations of Daikon's frontend. For MySQL, we replaced the default frontend with our own PIN tool. Daikon's backend generates invariants from the log dumped by our PIN tool. For Aget, a relatively small program, we manually inserted dummy functions before *every* global-variable read and then instructed Daikon's frontend kvasir-dtrace to record all global variables at the exit of these dummy functions. Note that this is not a fundamental limitation of Daikon. In fact, the commercial version of Daikon can provide invariants at instruction-level granularity, and could have been used straight out of the box.

MySQL1: After a training phase with a mix of 50 INSERT, 25 SELECT, and 25 DELETE queries to MySQL server, Daikon produced a total of 338 equality invariants, each associated with one instruction that reads a global variable. ConSeq considers these instructions as critical reads and detects that 13 out of these 338 invariants could be violated by pure interleaving changes. Among these 13, one of them points to a read of `binlog::log_type` inside function `MySQL::insert`. Daikon observes that this variable's value is always 3 (i.e., `LOG_OPEN`), while ConSeq finds that the value could become 0 (i.e., `LOG_CLOSED`) under an alternative interleaving. This turns out to be exactly the MySQL-1 bug. The abnormal `LOG_CLOSED` value would cause MySQL to miss some logging entries.

Among the other 12 possible violations reported by ConSeq, ten of them are false positives that cannot actually occur due to custom synchronization. The other two can truly occur and violate the candidate invariants proposed by Daikon. However, they are not bugs and do not lead to software failures.

Aget: Aget contains a concurrency bug that involves 11 threads and 21 shared variables (1 scalar and 20 entries in an array of structs), as shown in Figure 9. For purposes of illustration, we only show 11 involved variables here.



```
Thread i [1..10]                              Signal Thread
                                    S2: memcpy(dst.wthread,  wthread );
S1: wthread[i].offset += dw;
    lock ( l );
    ...
S3: bwritten += dw;
    unlock ( l );                   S4:  dst.bwritten =   bwritten  ;
                                          Aget Download.c, Resume.c
```

Figure 9. A multi-variable atomicity-violation bug that involves 11 threads and many shared variables.

After 50 training runs, Daikon generates equality invariants at 20 program locations. Based on that information, ConSeq reports 2 bugs. **The first bug is Aget1, which cannot be accurately detected by any previous tool. The second is a new bug in Aget that has never been reported.** Specifically, Daikon showed the following invariants just before S4 in Figure 9:

```
..dummy_bwritten_005():::EXIT
::bwritten == sum(::wthread[].bwritten)
```

It means the sum of each thread's `wthread[i].bwritten` fields (across all 10 worker threads) should be equal to `bwritten`, a global variable representing the total number of bytes that have been written by all threads. ConSeq then reports a suspicious interleaving (Figure 9) that could violate this invariant.

One point to note about the new Aget bug that ConSeq found is that it could be triggered by fewer than 11 threads (e.g., two worker threads and one signal thread would suffice). However, when the user does not specify the number of worker threads on the command line, the default is 10. We used the default in the Aget experiment to simulate an in-house tester who wishes to test a system's default configuration. Note that if a (user-supplied) test provokes a complicated situation in the initial run, that is what ConSeq must work with. It has no automatic way of "reducing" the run down to a minimal-size example. Fortunately, Daikon will work with any number of threads and does not have a major scaling problem; Daikon's work remains roughly the same, because each thread does less work when there are more threads. In summary, the Aget experiment illustrates the usage scenario of an in-house tester wishing to test a system's default configuration, and shows that ConSeq has the capability to detect bugs even when the input used generates a complicated interleaving scenario.

11. Related Work

Concurrency bug detection Sections 1 and 10 have already compared ConSeq with race detectors and atomicity-violation detectors. Recent work has studied multi-variable bugs [25, 31, 35, 63] and order problems [34]. In general, ConSeq complements these tools by starting from consequences rather than causes. ConSeq has a smaller false-positive rate than most traditional tools, and can effectively detect many concurrency bugs caused by complicated interleavings that are difficult for traditional tools to detect.

Recently, several invariant-based tools have been proposed for debugging concurrency bugs with complicated causes, such as definition-use violations and order violations [45, 55]. These tools rely on observing many training runs to identify abnormal interleavings. As a result, insufficient training will cause false positives for them. Furthermore, they cannot report bugs until the buggy interleavings are observed.

Another recent tool, ConMem [66], focuses on concurrency bugs that directly lead to memory errors, such as NULL-pointer dereferences on shared pointer variables. In terms of design philosophy, traditional detectors focus on certain interleaving patterns, which is one end of a spectrum of techniques that can be used to identify concurrency bugs whereas ConMem focuses on the other end (certain failure patterns). ConSeq takes matters even further: ConSeq starts from the failure and analyzes backwards through the error-propagation process. ConSeq is much more general than ConMem. It not only covers all major types of failure patterns (ConMem only covers one), but it also provides a general solution for bugs with various types of error propagation. ConMem does not consider the propagation process at all, and only considers buggy interleavings that directly cause memory bugs (i.e., the critical reads must be exactly the memory error sites)– a strategy that would be difficult to apply to other types of bugs. In terms of techniques, ConSeq is not a direct extension of previous work. To balance generality, analysis simplicity, and bug-report accuracy, ConSeq fully leverages the characteristics of concurrency bugs, and combines advanced static analyses with dynamic analyses.

Interleaving testing ConSeq shares the same goals of interleaving-testing tools [16, 39, 44, 53]. Such tools all try to explore the *interleaving* space for each *input* provided by developers during in-house testing. ConSeq uses synchronization analysis and perturbation-based interleaving enforcement in common with some of these tools [44]. The difference between ConSeq and previous tools is that ConSeq uses different methods to guide its exploration of the interleaving space.

RaceFuzzer [53] and CTrigger [44] use race/atomicity-violation detection results to guide their interleaving testing. ConSeq can complement them by exposing concurrency bugs that cannot be detected by race/atomicity detectors, as shown in Section 10. In terms of performance, the high false-positive rates of these bug-detection tools determine that RaceFuzzer and CTrigger will need to test many more interleavings than ConSeq for *each* input. On the other hand, ConSeq needs extra static analysis to identify critical reads. Fortunately, because ConSeq only needs to perform static analysis once for *all* inputs and *all* testing runs, the overhead will easily be offset by the interleaving testing time that is associated with *specific* inputs.

CHESS [39] guides its interleaving testing by bounding the number of preempting context switches. Although this is an effective heuristic, CHESS still faces the challenge of balancing coverage and performance, because its testing space increases polynomially in the number of potential context-switch points during the whole execution. If it considers each shared memory access to be a potential context-switch point, CHESS still cannot effectively explore the large interleaving space of applications like MySQL, as shown in previous work [58]. Therefore, CHESS often limits context switches only to synchronization points. This constraint will fail to expose many concurrency bugs, including many bugs discussed in this paper. Recently, people also proposed using randomized schedulers [7] to probabilistically expose concurrency bugs.

It works very well for simple concurrency bugs, such as many order-violation bugs, but will be ineffective for many other bugs in large applications, such as atomicity-violations or more complicated bugs.

In general, ConSeq and previous interleaving-testing tools complement each other by looking at the interleaving space from different perspectives.

Failure-diagnosis and repeating The consequence-oriented approach in ConSeq is similar in flavor to failure-diagnosis/replay tools [1, 15, 46, 58, 65]. For example, slicing is a useful technique in both cases [15, 58, 65]. The heuristic of paying more attention to the code close to a (potential) failure site is also widely used [46, 58, 65].

However, ConSeq and failure-diagnosis tools have different design purposes. ConSeq looks for unknown interleaving errors that can cause previously unobserved failures, while diagnosis tools try to repeat/diagnose a failure after it occurred. The different design purposes present different resource and scalability challenges.

Diagnosis tools come into play after a failure has occurred and been observed. Therefore, they usually assume complete knowledge about the failure core-dump, and sometimes also run-time information collected from the failure. They can spend a lot of resources on repeating and understanding one specific failure. On the contrary, ConSeq simply applies itself to every test input and does not know where failures might occur. It needs to prune out unlikely failure sites as quickly as possible, and expose hidden failures using limited testing resources. For example, in our experiments, ConSeq's trace analysis needs to triage hundreds of potential failure sites within seconds, while diagnosis tools frequently need a couple of hours or more to repeat one failure caused by a concurrency bug in a large application [1, 58], unless they have sufficient information from production runs [46]. Of course, ConSeq's scalability comes with a cost. ConSeq might miss some failures involving long propagation distances that diagnosis tools can successfully diagnose.

Input generation and symbolic execution Most failure-diagnosis tools assume full knowledge of bug-triggering inputs. ConSeq and other interleaving-testing tools [16, 39, 44, 53] also rely on input test-case generation to provide good test suites. Recently, symbolic execution [9, 20, 54, 65] has been used to generate high-coverage inputs for unit testing. DDT [26] and ESD [65] further extended this approach for concurrent programs. Using symbolic execution, DDT can detect synchronization problems caused by untimely interrupts in device drivers, and ESD can generate bug-triggering inputs and interleavings based on core-dumps from deadlock situations. These inspiring works can potentially help all testing and diagnosis tools, including ConSeq. However, due to the scalability constraints of symbolic execution and theorem provers, previous work only experimented with relatively small applications or relatively simple interleaving problems. Remember that exposing deadlocks is usually easier than exposing non-deadlock bugs, because it only needs to explore the different orders among lock acquisitions and releases, instead of the orders among all shared memory accesses. Further research is needed on how to scale symbolic execution to expose general concurrency bugs that go beyond simple lock-discipline issues; symbolic execution is currently not able to address concurrency bugs that are caused by arbitrary interleavings among arbitrary code segments in large, real-world applications.

Model-checking Model-checking research aims to verify properties of concurrent programs. Recently, a lot of progress has been made [27, 39, 48]. However, the state-explosion problem still exists. ConSeq has a different design goal and makes different trade-offs from model-checking tools. ConSeq does not try to obtain sound or complete results. ConSeq leverages the characteristics of concurrency bugs, especially the short-propagation heuristic, and uses a combination of static and dynamic analysis to make backward bug-detection feasible for large applications.

Run-time avoidance has also been studied for concurrency bugs. Dimmunix [24] learns lessons from previous deadlocks to avoid future deadlocks. Atom-Aid [36] and PSet [63] provide ways to survive concurrency bugs by prohibiting certain patterns of interleavings at run-time through hardware support. Deterministic execution [14] pushes this to an extreme. Software-only tools like Grace [4] and Kendo [43] achieve similar goals for certain types of multi-threaded programs at run-time. ConSeq complements such tasks by exposing concurrency bugs before they manifest in a production run.

12. Conclusions

This paper explores a backward, consequence-oriented approach to concurrency-bug detection. Our evaluation showed that ConSeq can detect 10 out of 11 tested concurrency bugs by monitoring correct testing runs. These bugs cover various types of failures and root causes, many of which cannot be detected by traditional race detectors and atomicity-violation detectors. ConSeq also found previously unknown bugs in open-source applications. Due to its consequence-oriented approach, ConSeq also has much better accuracy than traditional tools.

We believe ConSeq provides a nice complement to traditional concurrency-bug-detection tools. It demonstrates that we can leverage tools designed for sequential programs and sequential bugs, such as Daikon, to detect complicated concurrency bugs. Application developers can easily extend and adjust ConSeq by inserting sequential-style assertions and error messages in their code.

Of course, ConSeq still has limitations. First, similar to other dynamic bug-detection tools, ConSeq depends on test inputs for code coverage. Second, ConSeq currently focuses on the failure types detailed in this paper; while these failure types cover a large subset of all software failures, they are certainly not comprehensive. Third, ConSeq uses the short-propagation-distance heuristic; while this heuristic has proven to be effective in our rather inclusive benchmarks, it will inevitably cause some bugs that have long propagation distances to be missed by ConSeq.

Future work on ConSeq will concern several aspects. First, we plan to apply it to more applications and inputs. Second, we plan to extend it with more failure patterns. Third, we plan to try alternative distance metrics and use better customized synchronization-pruning techniques to further improve its accuracy.

13. Acknowledgments

We would like to thank our shepherd, Shimin Chen, and the anonymous reviewers for their invaluable feedback. We thank Bill Harris for many helpful discussions. We thank people from Daikon research group for providing much needed help. We also thank Mark Hill for his feedback and suggestions. Shan Lu is supported by a Claire Boothe Luce faculty fellowship, and her research group is supported by NSF grant CCF-1018180. Thomas Reps's research group is supported by NSF under grants CCF-0810053 and CCF-0904371, by ONR under grants N00014-09-1-0510 and N00014-10-M-0251, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, 2004.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[6] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, 1993.

[7] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[8] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.

[9] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[10] Cherokee. Cherokee: The Fastest free Web Server out there! http://www.cherokee-project.com/.

[11] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *EuroSys*, 2010.

[12] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[13] Click. The Click Modular Router Projec. http://read.cs.ucla.edu/click/click.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[15] M. Dimitrov and H. Zhou. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS*, 2009.

[16] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multi-threaded Java program test generation. *IBM Systems Journal*, 2002.

[17] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[19] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[21] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of Linux kernel behavior under errors. In *DSN*, 2003.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *TOPLAS*, 1990.

[23] R. Jhala and R. Majumdar. Software model checking. *Computing Surveys*, 41(4), 2009.

[24] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[25] N. Kidd, P. Lammich, T. Touilli, and T. Reps. A static technique for checking for multiple-variable data races. *Software Tools for Technology Transfer*, 2010.

[26] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX*, 2010.

[27] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 2009.

[28] I. Lee and R. K. Iyer. Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 Operating System. *IEEE*, pages 20–29, 1993.

[29] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[30] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA*, 2007.

[31] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, October 2007.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

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

[34] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[35] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA*, 2010.

[36] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, 2008.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[38] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[39] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[40] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[41] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[42] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[43] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

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

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

[46] 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*, 2009.

[47] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Incorporated, 1996.

[48] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[49] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies c a safe method to survive software failures. In *SOSP*, 2005.

[50] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: taming device drivers. In *EuroSys*, 2009.

[51] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[52] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

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

[54] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, 2005.

[55] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[56] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[57] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008.

[58] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.

[59] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, 1984.

[60] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[61] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, 2010.

[62] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[63] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[64] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[65] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys*, 2010.

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