ConAir: Featherweight Concurrency Bug Recovery Via Single-Threaded Idempotent Execution

Wei Zhang[1] Marc de Kruijf[1,2] Ang Li[1] Shan Lu[1] Karthikeyan Sankaralingam[1]

[1]Computer Sciences Department, University of Wisconsin–Madison [2]Google, Inc

{wzh,dekruijf,ali28,shanlu,karu}@cs.wisc.edu

Abstract

Many concurrency bugs are hidden in deployed software and cause severe failures for *end-users*. When they finally manifest and become known by *developers*, they are difficult to fix correctly. To support end-users, we need techniques that help software survive hidden concurrency bugs during production runs. To help developers, we need techniques that fix exposed concurrency bugs.

The state-of-the-art techniques on concurrency-bug fixing and survival only satisfy a subset of four important properties: compatibility, correctness, generality, and performance. We aim to develop a system that satisfies all of these four properties. To achieve this goal, we leverage two observations: (1) rolling back a *single* thread is sufficient to recover from most concurrency-bug failures; (2) reexecuting an *idempotent* region, which requires no memory-state checkpoint, is sufficient to recover from many concurrency-bug failures. Our system ConAir includes a static analysis component that automatically identifies potential failure sites, a static analysis component that automatically identifies the idempotent code regions around every failure site, and a code-transformation component that inserts rollback-recovery code around the identified idempotent regions.

We evaluated ConAir on 10 real-world concurrency bugs in widely used C/C++ open-source applications. These bugs cover different types of failure symptoms and root causes. Quantitatively, ConAir helps software survive failures caused by all of these bugs with negligible run-time overhead ($<1\%$) and short recovery time. Qualitatively, ConAir can help recover from failures caused by unknown bugs. It guarantees that program semantics remain unchanged and requires no change to operating systems or hardware.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.1 [*Operating Systems*]: Process Management

Keywords idempotency, concurrency bugs, failure recovery, static analysis, bug fixing

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

1. Introduction

1.1 Motivation

Concurrency bugs are unsynchronized memory accesses in multi-threaded programs. Many concurrency bugs are hidden in production-run software, causing severe failures in the field with huge financial losses [30, 42, 48]. When they finally get known by developers, correctly fixing them takes substantial manual effort. Developers often need weeks, or even months, to design a concurrency-bug patch [20, 35] yet about 40% of released concurency-bug patches are incorrect, which are the most error-prone among all types of bug patches [56]. Therefore, it is critical to help *end-users* enable production-run software to *survive* failures caused by hidden concurrency bugs and help *developers fix* known concurrency bugs.

An ideal bug fixing and survival technique should have several key properties: *compatibility*, i.e. no OS/hardware modification; *performance*, i.e. small run-time overhead and fast failure recovery; *generality*, i.e. helping bugs with a wide variety of root-cause interleaving patterns without reliance on accurate bug detection; *correctness*, i.e. not generating results infeasible for original software.

Table 1 summarizes three solutions to this problem and outlines their differences for all these four properties. As shown in columns 2, 3, and 4, no existing technique can achieve all four properties at the same time. We now elaborate on these techniques.

	Auto. Fixing	Prohibiting Interleaving	Rollback Recovery	ConAir
Compatibility	✓	*	*	✓
Correctness	✓	✓	✓	✓
Generality	-	*	✓	✓
Performance	✓	*	*	✓

Table 1. A comparison among concurrency-bug fixing and survival techniques (✓: yes; -: no; *: cannot all be yes at the same time.)

The *automatic fixing* approach statically or dynamically adds synchronization into programs to eliminate *known* bug-triggering interleavings [23, 24, 39]. Although promising, a tool with this approach only fixes bugs with a specific root cause (e.g., atomicity violations [23]), because it requires different types of synchronization to eliminate buggy interleavings of different root-causes. Furthermore, by design, this approach does not help software survive hidden bugs. It only fixes known bugs based on accurate bug root-cause detection.

Proactively prohibiting certain types of interleavings at run time is a common approach to surviving hidden concurrency bugs [5, 29, 37, 43, 45, 46, 51, 55, 57, 58]. Techniques based on this approach only survive failures caused by certain types of interleavings, and tend to impose unnecessary serialization and performance loss on existing hardware. In rare cases, some techniques belonging to this

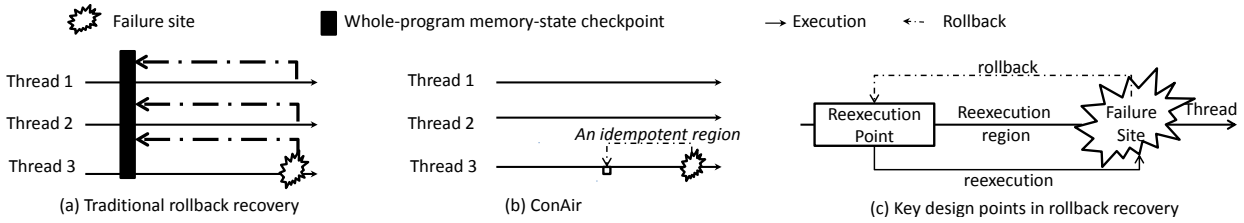

Figure 1. An overview of ConAir

approach may need programmer annotations to eliminate the effect of certain interleavings to maintain correctness.

The *rollback recovery* approach leverages the non-determinism of multi-threaded software to survive hidden concurrency bugs [44, 50, 53]. Without prohibiting any interleaving, it uses system checkpoint and rollback techniques to recover from failures, which is not limited to particular types of root causes. Unfortunately, existing techniques based on this approach require periodic *whole-program* checkpoint at run time and *whole-program* rollback for failure recovery. As a result, they require OS/hardware modifications to achieve good performance.

1.2 Contributions

This paper presents ConAir, a static tool that automatically inserts rollback-recovery code into multi-threaded software and allows software to recover from a wide variety of known and hidden concurrency bugs with little run-time overhead, as shown in Figure 1.

ConAir distinguishes itself from existing rollback-recovery systems by the following features:

1. **No multi-threaded rollback**. We observe that failures caused by most concurrency bugs can be recovered through rolling back just *one* thread, instead of all threads.
2. **No memory-state checkpointing**. We observe that failures caused by many concurrency bugs can be recovered by re-executing an *idempotent* region surrounding the failure site. A code region is idempotent if it can be reexecuted for any number of times without changing the program semantics. More formal definition of idempotent regions is in Section 2.2.

The above observations help ConAir achieve the four properties listed in Table 1. The reexecution of single-threaded idempotent regions guarantees no change to program semantics (*correctness*)[1]. The rollback-recovery approach, by design, allows ConAir to recover from concurrency bugs caused by a wide variety of root causes (*generality*). By avoiding the need for checkpointing memory state and by avoiding coordination across threads, ConAir limits its run-time overhead (*performance*) and requires no modification to OS/hardware (*compatibility*).

ConAir can be used in two modes. In survival mode, ConAir can be applied to harden a multi-threaded program against hidden concurrency bugs. In fix mode, it can generate safe temporary patches for concurrency bugs whose root causes are unknown. This is helpful to developers who often know the failure symptom of a reported bug long before they understand the root cause of the bug.

We have evaluated ConAir by using 10 real-world concurrency bugs in open-source server and client software. These 10 bugs represent bugs of common root causes, including atomicity violations, order violations, and deadlocks, and bugs of common failure symptoms, including assertion violations, wrong outputs, segmentation faults, and hangs. Without any knowledge of these bugs, ConAir automatically hardens the software at 7 – 19185 statically identified potential failure sites per program. The hardened software runs almost as fast as the original software, with only 0 – 0.2% run-time overhead. Failures caused by 8 out of 10 bugs can always be successfully recovered. The other 2 bugs lead to wrong-output failures. If the output-correctness conditions are known to ConAir, failures caused by these 2 bugs can also be successfully recovered. The time taken for failure recovery varies and is between 8 microseconds and 17 milliseconds. ConAir also has its limitations, which will be discussed in Section 6.5.

[1] ConAir does not violate memory consistency model (see Section 2.2).

2. ConAir overview

The design of a rollback-recovery system for multi-threaded software includes three key components, as shown in Figure 1c: (1) how many threads participate in the rollback recovery; (2) where the failure site is in each participant thread; (3) what is the reexecution point in each participant thread. We take a novel approach to these components by leveraging our key observations. To address component (1), *unlike* previous work, only one thread participates in ConAir rollback recovery. Components (2) and (3) are synergistically handled by forming idempotent code-regions whose end-point is the failure site, and start-point serves as a natural reexecution point for the participant thread. This section develops these key observations of ConAir in detail, and uses these observations to explain its *generality*, *correctness*, *performance*, and *compatibility* properties.

Terminology The program location where software fails is called a *failure site*. The location where software could fail is called a *potential failure site*. Since we may never know whether a failure would occur at a particular location, sometimes the paper does not differentiate failure sites and potential failure sites. A previous study shows that 97% of non-deadlock concurrency bugs first encounter their failures in a single thread, which may then terminate the whole program [61]. We refer to the thread that first encounters the failure as the *failing thread*.

2.1 Single-threaded rollback to recover concurrency bugs

We first discuss the *generality* of single-threaded recovery and compare it with traditional multi-threaded recovery. This discussion does not consider the *correctness* issue, which is discussed in Section 2.2. Our discussion and conclusions are based on root-cause and error-propagation characteristics of real-world concurrency bugs.

Recovering atomicity-violation bugs Atomicity violations contribute to about 70% of real-world non-deadlock bugs [35]. They occur when two code regions R_1 and R_2 from two threads interleave unserializably, which violates the expected atomicity of one or both regions. Clearly, if we can rollback and reexecute any one involved thread, the execution of R_1 and R_2 will be serialized and the failure will be recovered. Therefore, to understand whether single-threaded

```
/*Thread 1*/   /*Thread 2*/
log=CLOSE;     if(log!=OPEN)
log=OPEN;        {//output failure}
```

(a) Violating WAW atomicity (Rollback Thread 2 to recover)

```
/*Thread 1*/   /*Thread 2*/
ptr=aptr;
tmp=*ptr;      ptr=NULL;
```

(b) Violating RAW atomicity (Rollback Thread 1 to recover)

```
/*Thread 1*/  /*Thread 2*/
if(ptr)
  fputs(ptr);  ptr=NULL;
```

(c) Violating RAR atomicity (Rollback Thd 1 to recover)

```
/*Thread 1*/                          /*Thread 2*/
cnt+=deposit1;
printf("Balance=%d",cnt);   cnt+=deposit2;
```

(d) Violating WAR atomicity (Rollback Thread 1 to recover)

Figure 2. Most failures caused by atomicity violations can be recovered by rolling back one thread, the failing thread (Different checkpoint/sandbox techniques may be needed to guarantee correctness.).

recovery works, we need to know whether the failing thread is involved in the unserializable interleaving.

We checked 51 real-world atomicity-violation bugs collected by a previous work [35]. About 92% of them cause failures in a thread that is involved in the unserializable interleaving and hence can potentially be recovered by single-threaded recovery.

The above observation can be better understood through bug examples shown in Figure 2. This figure depicts the most common types of real-world atomicity violations [33, 35, 52]. As we can see, an atomicity violation usually causes an involved thread to read an unexpected value from a shared variable, such as log in Figure 2a, ptr in Figure 2b and Figure 2c, and cnt in Figure 2d. This incorrect value quickly leads to a failure in that thread. Clearly, the failure can be recovered by rolling back and reexecuting that thread.

Recovering order-violation bugs Order violations contribute to nearly 30% of real-world non-deadlock concurrency bugs [35]. They occur when an operation A is expected to execute before an operation B, but instead executes after B due to lack of synchronization. Clearly, if we can rollback and reexecute the thread of B, the occurrence of B will be effectively delayed and the failure will be recovered. Since single-threaded recovery always rolls back the failing thread, we need to understand whether the thread of B is the failing thread.

We checked all 21 real-world order-violation bugs collected by a previous work [35]. We found that about 50%of order-violation bugs lead to failures in the thread of B, and hence can be recovered by the single-threaded recovery. Failures of the other bugs manifest in the thread of A and occasionally some other threads.

To better understand this observation, one can consider a common type of order-violation bugs: thread t_B reads a shared-variable V before V is initialized by thread t_A. In this case, the uninitialized value in V usually leads to a failure in t_B. By rolling back t_B, we can postpone the read of V until V is initialized.

Recovering deadlock bugs Deadlock contributes to about 40% of all concurrency bugs [31]. When a deadlock occurs, every thread involved is holding some resource that is blocking another thread. In a typical deadlock, making any thread release a resource will break the circular resource dependence. Clearly, rolling back any single failing thread can recover from a deadlock.

Summary Overall, rolling back a single thread (i.e., the failing thread) is effective to recover from most concurrency bugs: all deadlocks, almost all atomicity violations (47 among the 51 studied), and about half of the order violations (11 among the 21 studied).

2.2 Idempotent reexecution to recover concurrency bugs

In general, an idempotent region is a code region that can be reexecuted for any number of times without changing the program semantics. Figure 3 shows a code-snippet that is idempotent contrasted with one that is not. In general, such regions can have arbitrary start and end-points in the program, so long as the code in that region adheres to idempotency semantics.

We narrow the definition to make the regions amenable for bug recovery. In this work, an idempotent region always ends at a potential failure site. It does not contain any writes to shared

```
y=x+1;
z=x+y;
```

(a) An idempotent region

```
x=x+1;
z=x+y;
```

(b) Not an idempotent region

Figure 3. Idempotency

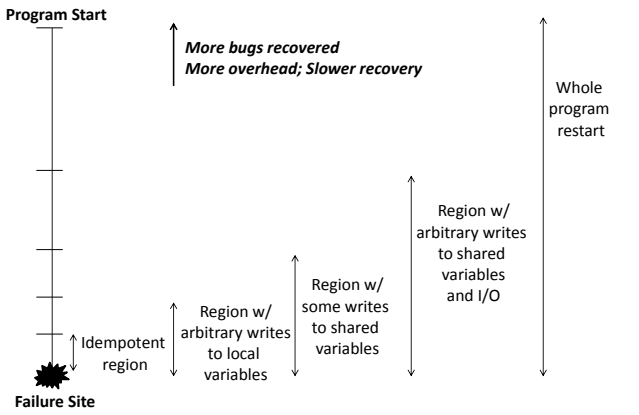

Figure 4. The tradeoff of reexecution-region design

variables, so that its single-threaded reexecution does not violate the memory-consistency model of hardware and system. It does not contain any I/O operations. It also does not contain any writes to local variables that could cause incorrect reexecution.

Using idempotent regions as reexecution regions can easily achieve *correctness* and good *performance*, because they can be correctly reexecuted without any checkpointing or logging. The only concern is that they may be too short to recover concurrency bugs. On the other hand, longer reexecution regions can recover more bugs, but also hurt *performance* and/or *compatibility* more, because they require more complicated rollback-reexecution techniques, such as sandboxing I/O events, buffering shared-variable modifications, and checkpointing local-variable values. Figure 4 sketches this trade-off and design space.

Fortunately, several empirical observations indicate that constraining reexecution regions to be free of idempotency-destroying operations does not eliminate the chance of recovering from concurrency-bug failures.

- I/O operations. Idempotent regions cannot contain any I/O operations. A study of real-world concurrency bugs shows that only about 15% of concurrency bugs' recovery involves I/O operations [54].
- Shared-variable writes. Idempotent regions cannot contain any write to shared variables. Fortunately, among all types of common concurrency bugs discussed earlier, only RAW and WAR atomicity violations (Figure 2b and Figure 2d) require reexecuting shared-variable writes to recover.

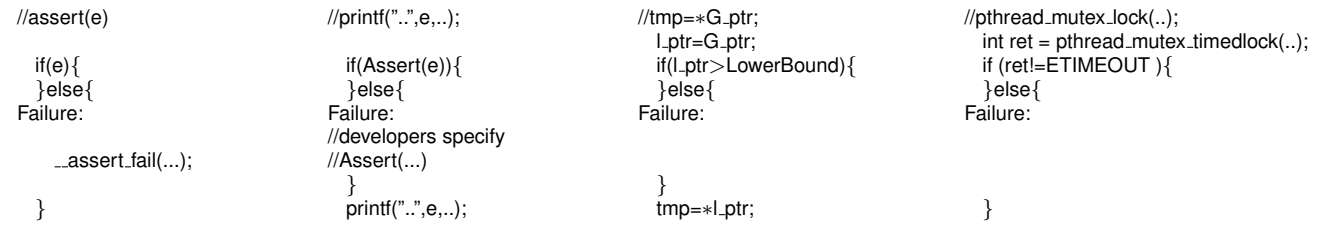

Figure 5. Failure sites for different types of failures (Some of them involve ConAir code transformation; LowerBound is 10,000 by default.)

- Local-variable writes that are not idempotent, such as the write to x in Figure 3b. No previous study has looked at this. Fortunately, one study of real-world concurrency bugs shows that most non-deadlock bugs have short error-propagation distances, often a handful of data/control dependence edges within one thread [61]. It is likely that many bugs' error-propagation does not involve such idempotency-destroying writes. As for deadlock bugs, they can be recovered by rolling back *any* single involved thread. It is likely that at least one thread can release a deadlock-inducing resource when rolling back its idempotent reexecution region[2].

To know the exact percentage of real-world failures that require each type of reexecution regions is difficult — most of the real-world concurrency bugs examined in previous work [35] have never been reproduced in a research environment. Therefore, we studied all the 26 bugs repeated and presented by 6 recently published works on concurrency-bug detection and prevention [23, 24, 49, 54, 60, 61]. Among these 26 bugs, 20 can be survived through single-threaded reexecution[3]. Among the reexecution regions of these 20 bugs, 16 are idempotent, 2 contain I/O operations, and 2 contain non-idempotent memory writes but no I/Os. Section 6 will present real-world failure examples that can be recovered by reexecuting idempotent regions in the failing thread.

Summary Traditional techniques mainly trend toward the right end of the reexecution-region design spectrum in Figure 4. Their focus of the failure-recovery universality inevitably leads to large run-time overhead and complicated/non-existing platform support.

This paper will explore the leftmost end of the design spectrum. We use idempotent regions as reexecution regions, and identify a reexecution point as the starting point of the idempotent region surrounding each failure site. Our design does not aim the universality of failure recovery. Instead, it aims to survive a significant portion of concurrency-bug failures with a wide variety of root causes at negligible overhead on existing platforms, which will allow easy adoption in production systems.

In the following, Section 3 presents a basic design and implementation of ConAir. Section 4 discusses further extensions and optimizations of ConAir, such as how to avoid useless recovery attempts and how to conduct inter-procedural recovery. Section 5 and Section 6 present the evaluation of ConAir.

3. ConAir design and implementation

ConAir framework includes three components:

1. A static analysis component that identifies failure sites in software (Section 3.1).
2. A static analysis component that identifies reexecution points for every failure site (Section 3.2).
3. A static code-transformation component that enables a multi-threaded program to survive concurrency bugs at the failure sites identified above through single-threaded rollback (Section 3.3).

ConAir does not aim to handle all possible software failures. Instead, it aims to handle common failures with a variety of failure symptoms and root causes with good run-time performance and no modification to the OS or hardware. ConAir also provides guarantee to never deviate from the original software semantics.

3.1 Failure site identification

Failure sites are where failures occur. Some failures may occur due to hidden bugs and some failures may have already manifested with their symptoms known to users/developers. To handle these two types of failures, ConAir operates in two modes: survival mode and fix mode. These two modes only differ in how the failure sites are identified.

3.1.1 Identifying failure sites in survival mode

Without any knowledge of hidden concurrency bugs, ConAir uses static analysis to identify program locations where common failures could occur. The following four types of failures are the most common among the real-world concurrency bugs [61].

Assertion failures. The assert macro is widely used by developers to specify critical program properties. In Linux systems, an assertion failure will cause the execution of __assert_fail(...). ConAir identifies the invocation of __assert_fail(...) as a (potential) failure site, as shown in Figure 5a.

Wrong outputs. Wrong output failures occur when software produces an incorrect output or fails to produce any output when an output is desired. Judging a wrong-output failure requires oracles specified by developers or users. The current prototype of ConAir can help recover from wrong-output failures, if developers can provide output oracles in the format of assert as shown in Figure 5b.

Segmentation-fault failures. A previous study [60] shows that most segmentation faults caused by concurrency bugs occur during the dereference of a heap/global pointer variable. Therefore, ConAir identifies every dereference of a heap/global pointer variable as a potential segmentation fault failure site, as shown in Figure 5c.

Deadlock failures. There are different ways to detect a deadlock failure. Some previous work [24] instruments Pthread library functions and reports deadlocks by catching cycles in the run-time resource-acquisition graph. Many real-world multi-threaded systems, such as MySQL [40], simply maintain a timer for each lock acquisition function and report a deadlock once the lock-acquisition times out. ConAir can work with any deadlock-detection mechanism: the detection code that reports a deadlock is treated as a (potential) failure site. Our current prototype assumes the time-out based

[2] Strictly speaking, idempotent code regions cannot contain lock functions. ConAir's techniques to solve this problem are discussed in Section 4.1.

[3] The 20/26 single-threaded recovery rate is lower than that among a larger set of real-world bugs presented in Section 2.1, because some papers [49, 60] use disproportionally large numbers of order-violation benchmarks.

deadlock detection. ConAir transforms every pthread_mutex_lock function into pthread_mutex_timedlock, and identifies failure sites accordingly as shown in Figure 5d. ConAir can handle customized lock functions, as long as the developers specify the prototypes of their lock, unlock, and timeout-lock functions.

ConAir does not require its failure-site identification to be sound or complete. Inevitably, many sites identified above never manifest as failures. Treating them as potential failure sites only causes negligible run-time overhead, as we will see in Section 6.2, benefiting from ConAir's low-overhead design. The above analysis can be easily customized to cover more types of failures or to focus on a smaller set of severe failures.

3.1.2 Identifying failure sites in fix mode

Fix mode can be used when users or developers encounter a non-deterministic failure with an *unknown* root cause. In this case, users or developers inform ConAir of the failure location. For example, when the bug shown in Figure 2b manifests, users or developers will observe a segmentation fault at the statement tmp=∗ptr, which ConAir treats as the failure site.

3.2 Reexecution point identification

As discussed in Section 2.2, the placement of reexecution points and reexecution regions largely determines the system performance. ConAir uses idempotent regions as its reexecution regions during failure recovery. Each reexecution point is the starting point of an idempotent region, which ends at a potential failure site. This design makes ConAir lightweight and able to recover from many, although not all, concurrency-bug failures.

3.2.1 Principle of identifying idempotent regions

Identifying idempotent code regions is not trivial. A code region that is *not* idempotent in source code, such as x=x+1, could become idempotent in bitcode, such as $x_1=x_0+1$, due to variable renaming conducted by a compiler. A code region that is idempotent in bitcode could later become *not* idempotent in binary code due to physical-register allocation. Due to these challenges, there are usually two approaches to identifying idempotent code regions in the binary code. One is to rely on binary code analysis alone. Unfortunately, this could be very complicated for x86 code. The second approach, which is used by recent work [12], is to use a combination of bitcode/binary-code analysis and bitcode/binary-code transformation.

ConAir takes the second approach using the LLVM static analysis and code generation framework [27]. As discussed in Section 2.2, an idempotent region does not contain shared-variable writes, non-idempotent local-variable writes, or I/O operations. Following this, ConAir identifies an idempotent region as an LLVM bitcode region that contains none of the following *idempotency-destroying instructions*: (1) writes to global or heap variables; (2) writes to local variables that are not allocated in virtual registers[4]; (3) function-call instructions. This code region is guaranteed to be idempotent at bitcode level.

To guarantee the region is also idempotent in the binary code, ConAir performs two transformations. First, ConAir uses the −no−stack−slot−sharing flag for LLVM to generate the binary code. This flag guarantees that different virtual registers, when not allocated in physical registers, are allocated in different stack slots. Under this configuration, the code regions identified above will always conduct idempotent operations on memory states. The only concern is that these regions may modify the value of a physical register and cause the reexecution to read a different register value from the original execution. Therefore, ConAir saves the register image at the beginning of the code region and restores the register image right before a rollback. The register save and restore are conducted by setjmp and longjmp. They are both very lightweight, taking only a few nanoseconds.

Alternative methods to identify idempotent code regions Some code regions that contain *idempotent-destroying operations* are still idempotent in binary code. For example, writing a stack variable v that is not allocated in virtual registers does not necessarily hurt the idempotency of a code region R, unless this write is preceded by a read of v that is not preceded by another write to v. As another example, some function calls do not hurt the idempotency. With more complicated analysis, we could identify more and longer idempotent regions in the future.

An alternative implementation decision is to modify the register allocator. A recent work [12] first identifies the boundaries of idempotent regions in LLVM bitcode, it then modifies the compiler back-end code generator to guarantee that idempotent bitcode is translated to idempotent binary code [12]. For our work, we took the setjmp/longjmp approach because it is easier to implement and is ISA independent. A production use of ConAir could employ either approach. The previous work [12] also splits the whole program into idempotent code regions, covering every instruction by idempotent regions. In contrast, our work only identifies idempotent regions that end at potential failure sites. This allows us to achieve negligible overhead ($< 1\%$) in our experiment (Section 6). On the contrary, previous work [12] could have more than 10% run-time overhead.

3.2.2 Algorithm of identifying idempotent regions

When a program does not contain any branch instruction, identifying reexecution points is straightforward. For every failure site f, we simply need to analyze statements one by one backwardly until we find the first statement s that is an *idempotency-destroying* instruction. The reexecution point is right after s.

Unfortunately, real programs always contain branch instructions and there could be multiple execution paths leading to a failure site f. Therefore, we have to identify an appropriate reexecution point along every path leading to f.

ConAir conducts a backward depth-first search from f. This static analysis starts with pushing the predecessors of f in the control-flow graph (CFG) into a work-list stack, and keeps processing the top statement in this stack as follows. (1) When the analysis encounters an idempotency-destroying operation, ConAir identifies a reexecution point right after this operation. ConAir then removes this statement from its work list. (2) When encountering the entrance of function containing f, ConAir identifies it as a reexecution point and removes it from its work list. This decision means that ConAir reexecution does not touch the caller of f. We will revisit this decision and discuss inter-procedural recovery in Section 4. (3) When encountering other statements, ConAir checks how many predecessors of this statement have not been visited. If there is none, ConAir removes this statement from the work list. Otherwise, ConAir pushes an unvisited predecessor of this statement to the top of its work list. ConAir stops its analysis when its work list is empty. At that point, all reexecution points for f are identified. The complexity of this analysis is linear to the static function size.

ConAir repeats the above algorithm for every failure site. Note that the reexecution points of different failure sites do not conflict with each other. That is, the reexecution region of a failure site f_1 will never get shortened by the reexecution points of another failure site f_2. The reason is that a reexecution point is always right after an idempotency-destroying operation or at the entrance of a function, which is the same for all failure sites.

[4] In LLVM, a virtual register is a variable in static single assignment form (SSA) [8]. It is statically assigned only once.

```
1
2
3
4
5
6
7
8  if(e){
9  }else{
10
11
12
13
14   __assert_fail(..);
15 }
```

(a) Original code

```
1   __thread jmp_buf c;
2   __thread int RetryCnt=0;
3   ...
4  Reexecution:
5   setjmp(c);
6   ...
7   //reexecution region
8   if(e){
9   }else
10 Failure:
11    while(RetryCnt++<maxRetryNum){
12      longjmp(c,0);
13    }
14    __assert_fail(..);
15  }
```

(b) Transformed code

Figure 6. ConAir code transformation for assert(e)

3.3 Transformation at failure sites and reexecution points

After identifying failure sites and reexecution points, ConAir performs the following code-transformations that enable the multi-threaded software to recover from concurrency-bug failures.

At every reexecution point, ConAir inserts a setjmp (line 5 in Figure 6) to make sure our reexecution region is idempotent. Sometimes, multiple failure sites may share a common reexecution point. In these cases, ConAir makes sure to insert just one, instead of multiple, setjmp at the common reexecution point.

At every failure site, ConAir inserts a longjmp to rollback the execution to the reexecution point (line 12 in Figure 6). While restoring the register image c, longjmp automatically changes the program counter to the reexecution point where c was taken. This naturally accomplishes the control-flow rollback. ConAir supports multiple reexecutions through the loop on Line 11, because some failures may require several rounds of reexecution to recover. The loop-condition variable is a configurable threshold that prevents endless recovery attempts. Its default value is one million.

ConAir uses a *thread-local* variable to save the register image at every reexecution point (line 1 in Figure 6). At run time, this variable always keeps the register image taken at the most recent reexecution point in a thread. This guarantees that the program will roll back to the right reexecution point.

Deadlock recovery can potentially lead to a livelock problem. This occurs when multiple threads involved in the deadlock try to rollback at exactly the same time. This issue can be solved by putting a small random sleep at the failure site.

3.4 Discussion

Future work can extend ConAir by extending its failure-site identification. Some potential failure sites could be pruned, if we can statically prove that failures can never occur there. For example, analysis could know that NULL-pointer dereference may never occur at some places [6]. We can also use dynamic technique like ConSeq [61] to prune well tested potential failure sites. We can also enlarge the set of potential failure sites based on developers' annotations or automatically inferred specifications. For example, ConAir currently inserts an assertion before every fputs function call to check whether the parameter of fputs is NULL or not.

Future work can also explore other designs of the reexecution regions. For example, some regions that write shared variables can be correctly reexecuted with more sophisticated rollback or checkpoint techniques.

4. Optimizations and Extensions

This section discusses how we extend the basic design of ConAir to recover from more concurrency bugs and optimize the basic design to achieve better performance.

4.1 Extending reexecution regions for library functions

The basic design of ConAir reexecution regions is very stringent: it cannot contain any function calls. In the following, we extend the basic reexecution regions to include some library-function calls.

Why do we need to reexecute library functions? Some failures cannot be recovered unless some library-function calls are reexecuted. For example, if we do not allow a reexecution region to call pthread_mutex_timedlock, ConAir can never recover a deadlock failure shown in Figure 5d. In fact, a reexecution region has to include a call of pthread_mutex_lock to recover deadlock failures, which we will discuss later.

Which library functions can be correctly reexecuted? Some library functions can be correctly reexecuted by executing *compensation functions* at the failure site. For example, if a code region executes a malloc, we need to call a free at the failure site; if a code region executes a pthread_mutex_lock, we need to call a pthread_mutex_unlock at the failure site.

Some library functions cannot be easily reexecuted. For example, output functions in general are difficult to reexecute without system support. Reexecuting free or pthread_mutex_unlock could also be dangerous. Imagine a code region that frees an object that is allocated before this region starts or releases a lock that is acquired before this region starts. It is almost impossible to correctly reexecute this type of code regions.

Implementing library-function extension in ConAir Following the above observations, we allow ConAir reexecution regions to contain memory-allocation functions and lock functions. Other functions, such as free, unlock, and output functions, are still considered idempotency-destroying and cannot be included in any reexecution region.

To support this extension, three changes are made. First, ConAir instruments every call site of memory-allocation and lock functions to record which region is allocated and which lock is acquired. ConAir checks the return value of pthread_mutex_timedlock to know whether a lock is acquired. Second, ConAir needs to know which memory-regions/locks are acquired in the *current* reexecution region at a failure site. To support this, ConAir maintains an integer counter for each thread, which is increased by one at every reexecution point. At the return of every memory-allocation/lock function, ConAir stores the starting address of the newly allocated region or the address of the newly acquired lock, as well as the current counter value of this thread, into a per-thread vector maintained by ConAir. Before storing the new record, ConAir cleans the vector, if the current counter value is different from what is stored in the vector. Third, at each failure site, before the longjmp, ConAir inserts code to iterate through the vector, identify every region/lock that is allocated/acquired under the current counter value, and deallocate/free it.

The above extension guarantees the recovery correctness. Note that deallocating a memory region R or releasing a lock L at the failure site in thread t_1 does *not* affect the correctness of other threads. Since a reexecution region cannot contain writes to shared variables, other threads could not have obtained any pointer pointing to R. Furthermore, no other thread could have acquired L before t_1 releases it. Also note that reexecution regions do not contain any free or pthread_mutex_unlock functions. Therefore, we do not need to worry about an object/lock that is allocated/acquired and then freed/released during one reexecution region.

Reexecution:	Reexecution:	Reexecution:	Reexecution:
lock(&L);//blocked	**lock**(&L0); **lock**(&L); //blocked	tmp=tmp+1; assert(tmp);//violated	tmp=global_x; assert(tmp);//violated
(a) Can**not** be recovered	**(b)** Could be recovered	**(c)** Can**not** be recovered	**(d)** Could be recovered

Figure 7. Some failure sites cannot be recovered by ConAir (The last line in each figure is a potential failure site)

4.2 Optimizations to remove unnecessary rollbacks

Some failures cannot be recovered by ConAir, such as those shown in Figure 7a and Figure 7c. We identify failure sites that are statically proven to be unrecoverable and remove any unnecessary rollback-reexecution code inserted by the basic ConAir algorithm described in Section 3.

Deadlock failure optimizations To recover from a deadlock failure, ConAir should at least release a lock originally held by the thread at the failure site, as shown in Figure 7b. Otherwise, other threads involved in this deadlock cannot possibly make progress during the recovery attempt, and hence ConAir has no chance to recover the deadlock. ConAir optimization follows this intuition. Each deadlock failure site f could correspond to different reexecution regions along different execution paths. ConAir checks whether there is a lock-acquisition operation inside at least one reexecution region of f. If there is none, no lock will be released at f and there is no chance for deadlock recovery. Therefore, ConAir removes the failure-recovery code at f. The current prototype of ConAir turns pthread_mutex_lock functions into pthread_mutex_timedlock functions when it identifies potential deadlock sites, as discussed in Section 3.1.1. Once ConAir identifies a failure site to be unrecoverable, the corresponding pthread_mutex_timedlock is turned back to pthread_mutex_lock.

Non-deadlock failure optimizations To recover from a non-deadlock failure, the reexecution conducted by ConAir should include at least one shared-variable read that can affect the evaluation outcome at the failure site, as shown in Figure 7d. Otherwise, the reexecution is guaranteed to fail again. Following this intuition, ConAir checks every non-deadlock failure site f. ConAir first uses intra-procedural static backward slicing to identify global/heap memory-read instructions that can affect f through data dependence and/or control dependence. ConAir then checks whether there is at least one such read instruction that is inside a reexecution region of f. If not, f is not recoverable and no failure-recovery code is inserted for it.

Our intra-procedural backward-slicing analysis is implemented in LLVM to analyze LLVM bitcode. Interestingly, our analysis is much simpler than general backward-slicing algorithms.

A major source of complexity in general slicing analysis is tracking data dependence through memory accesses, which requires pointers-alias analysis, as shown by the dotted line in Figure 8a.

ConAir does *not* have this concern. Recall that write instructions in a ConAir reexecution region only write to virtual registers (Section 3.2), such as the write to %0, %1, and %2 in Figure 8b. Therefore, when ConAir backward slicing encounters a read instruction r that does not read from a virtual register, such as line 2 in Figure 8b, ConAir simply *stops* tracking its data dependence. The reason is that the instruction that provides value for r must write to non-virtual-register locations (e.g., line 1 in Figure 8b) and do not belong to ConAir idempotent regions. Slicing outside an idempotent region, and hence a reexecution region, is useless for ConAir.

After removing all unrecoverable failure sites, ConAir also removes reexecution points that do not correspond to any failure site and finishes the optimization.

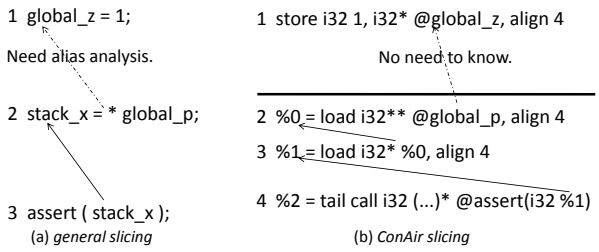

Figure 8. The difference between general slicing analysis and the slicing in ConAir (the stack_ prefix denotes stack variables; the global_ prefix denotes global variables; a solid arrow shows an easy-to-get dependence; a dotted arrow shows a difficult-to-get dependence; the thick line in (b) shows the reexecution-region boundary.)

4.3 Inter-procedural reexecution

The basic ConAir algorithm presented in Section 3 only attempts intra-procedural recovery. That is, the reexecution point for a failure site f is always in the function that contains f, referred to as foo. In this section, we discuss pushing reexecution points into the callers of f. We refer to this as *inter-procedural recovery*. Inter-procedural recovery can help recover from more failures, but can also hurt performance.

When is inter-procedural recovery correct? We should not attempt an inter-procedural recovery, if doing so would generate results infeasible for the original program. Following the discussion in Section 3.2, when there is no idempotency-destroying operation on a path between f and the entrance of foo, it is safe to extend the reexecution region into the caller of foo.

When can inter-procedural recovery help? Since a ConAir reexecution region cannot contain any modification to shared variables, parameters are the only ways for a caller to affect the execution outcome at f. Therefore, inter-procedural reexecution can potentially help the recovery of a non-deadlock failure, only when a parameter of foo is on the backward slice of f.

When do we need inter-procedural recovery the most? It is difficult to accurately predict when inter-procedural recovery will be needed without knowing the hidden bugs. Intuitively, imagine a path p between the entrance of foo and f. As discussed in Section 4.2, reexecuting p cannot help recover a non-deadlock failure at f, if p contains no shared-variable read that can affect the outcome of f. Similarly, reexecuting p cannot help recover a deadlock failure at f, if p contains no lock-acquisition functions. Therefore, we hypothesize that an inter-procedural recovery is most needed when such an *unrecoverable* path p exists.

Conditions of ConAir inter-procedural recovery Inter-procedural recovery could help recover more failures. However, it significantly slows down ConAir static analysis, which we will see in Section 6; it will also identify more reexecution points with more setjmp executed at run time, which incurs more overhead.

Based on these considerations, ConAir selects a failure site f for inter-procedural recovery when f satisfies all the following three

conditions: (1) There is no idempotent-destroying operation on any path between the entrance of foo and f. This way, once selected for inter-procedural recovery, the recovery attempt of f is always conducted interprocedurally, no matter which path is followed in foo during the failure run. (2) At least one argument of foo is on the backward slice of f, when f is a non-deadlock failure site. This way, the inter-procedural reexecution can potentially help recover the failure. This parameter is referred to as a *critical parameter*. (3) At least one path between the entrance of foo and f is unrecoverable.

How to identify inter-procedural reexecution point? When a failure site f is identified for inter-procedural recovery, ConAir uses static analysis to find every function foo_1 that calls foo. Inside foo_1, we use the analysis described in Section 3.2.2 to look for reexecution points. This analysis starts from the instruction that pushes the critical parameter onto the stack of foo, when f is a non-deadlock site; it starts from the invocation of foo, when f is a deadlock site.

We then identify reexecution points just as during intra-procedural recovery (Section 3.2). Note that the setjmp and longjmp inserted at reexecution points and failure sites handle the program-counter register and the stack-frame registers. Therefore, no extra effort is needed for inter-procedural rollback.

While analyzing the caller of foo, we could decide to try inter-procedural recovery again. In our current prototype, we set a threshold of how many levels of inter-procedural recovery we would attempt for one initial failure site f. The default setting is 3. That is, to recover f inside a function foo, ConAir could at most rollback the execution to the callers' callers' caller of foo, referred to as foo_3. This threshold is configurable. It balances the recovery capability and run-time performance.

Theoretically, there could be a path between the entrance of foo_3 and f that does not contain any idempotent-destroying operations. Since we decide not to go further into the caller of foo_3, we could choose to set the reexecution point at the entrance of foo_3. However, this scheme could prevent failure sites inside foo_3 to attempt inter-procedural recovery. Therefore, in our current prototype, ConAir simply gives up the inter-procedural recovery attempt of f in that case and puts the reexecution point back to the entrance of foo. Note that, this case is extremely rare and has *never* occurred in any of the applications evaluated by us.

Other issues Our inter-procedural recovery analysis needs to work together with the intra-procedural recovery analysis discussed in Section 3 and the optimization analysis discussed in Section 4.2. ConAir first conducts intra-procedural analysis. This analysis could identify the entrance of foo as a reexecution point for a failure site f, referred to as RE_{intra}. ConAir then conducts inter-procedural recovery analysis. Once f is identified for inter-procedural recovery, ConAir safely removes the reexecution point RE_{intra}[5]. Finally, ConAir conducts its optimization analysis discussed in Section 4.2. This optimization is *only* applied to failure sites that conduct intra-procedural recovery. Failure sites that are selected for inter-procedural recovery usually have long reexecution regions. It is much harder to statically prove them to be unrecoverable.

5. Experimental Methodology

Our work aims to allow programs to recover from a significant portion of real-world concurrency-bug failures with a wide variety of root causes at low overhead on existing platforms. To empirically evaluate whether ConAir has achieved this goal, our experiments look at 10 real-world concurrency-bug failures in 10 open-source applications that have been widely used in previous bug detection and avoidance research [23, 24, 49, 54, 60, 61]. They represent a wide variety of failure symptoms and root causes, as shown in Table 2. We will quantitatively evaluate whether ConAir can indeed recover failures with different root causes; what is the run-time overhead introduced by ConAir; how long it takes to recover a failure under ConAir; and the static-analysis complexity of ConAir.

[5] This could cause some other failure sites in foo to conduct interprocedural recovery too, which is fine.

App.	App. Type	LOC	Failures	Causes
FFT	Scientific computing	1.2K	w. output	A/O Vio.#
HawkNL	Network library	10K	hang	deadlock
HTTrack	Web crawler	55K	seg. fault	O Vio.
MozillaXP	XPCOM: cross platform component object model	112K	seg. fault	O Vio.
MozillaJS	JavaScript engine	120K	hang	deadlock
MySQL1	Database server	681K	w. output	A Vio.
MySQL2	Database server	693K	assertion	A Vio.
Transmission	BitTorrent client	95K	assertion	O Vio.
SQLite	Database engine	67K	hang	deadlock
ZSNES	Game simulator	37K	assertion	O vio.

Table 2. Applications and Bugs (w. output: wrong output failures; A Vio.: atomicity violations; O Vio.: order violations; #: There are both order violations and atomicity violations in FFT.)

We apply ConAir to analyze and transform each application twice, representing fix mode and survival mode respectively.

While applying ConAir in survival mode, ConAir needs **no** knowledge of failures or bugs. It automatically identifies potential failure sites as discussed in Section 3.1 and transforms software.

While applying ConAir in fix mode, ConAir assumes the knowledge of failure sites provided by developers or users who want to fix a particular failure they observed. This could be a specific assert that is violated; a particular pthread_mutex_lock that blocks the program; a particular memory-access instruction that causes a segmentation fault; or an output function that generates incorrect results. Note that ConAir needs **no** information about bug-triggering inputs, bug root-causes, or bug-detection results.

To evaluate whether the software can survive the manifestation of a bug, we insert sleeps into each program's buggy code regions to force the occurrence of the failure-inducing interleaving. Executed under this setting and failure-inducing inputs, the software in our benchmark set fails with almost 100% probability, if ConAir is not applied. After ConAir is applied, we execute the software under the same setting for 1000 times. We claim ConAir to have successfully recovered the failure if the hardened software executes correctly in all 1000 runs. To evaluate the run-time overhead, we execute the original program and the transformed programs under the same input (i.e., the bug-triggering input) for 20 times each, and calculate the average overhead. No sleep is inserted and software never fails during the run-time overhead measurement.

Among all types of failures, wrong-output failures cannot be recovered unless the users or developers annotate the correctness condition of an output. This condition is easily available in fix mode, but is not necessarily available in survival mode. To better understand the *worst*-case overhead of ConAir in the survival mode, ConAir treats every output function as a potential failure site, even though the correctness condition may be unavailable.

All the experiments are conducted on an 8-core Intel Xeon machine running Linux version 2.6.18 and using the LLVM 2.8 compiler.

6. Experimental Results

As shown in Table 3, ConAir recovers all the evaluated bugs. ConAir incurs no overhead in fix mode, and negligible overhead ($< 1\%$) in survival mode. In this section, we will explain the following experimental results in detail: (1) how ConAir effectively fixes bugs with known failure symptoms; (2) how ConAir transparently hardens

App.	Failure Recovered?		Overhead	
	fix	survival	fix	survival
FFT	✓	$✓_c$	0%	0.0%
HawkNL	✓	✓	0%	0.0%
HTTrack	✓	✓	0%	0.0%
MozillaXP	✓	✓	0%	0.0%
MozillaJS	✓	✓	0%	0.0%
MYSQL1	✓	$✓_c$	0%	0.1%
MYSQL2	✓	✓	0%	0.0%
SQLite	✓	✓	0%	0.0%
Transmission	✓	✓	0%	0.2%
ZSNES	✓	✓	0%	0.0%

Table 3. Overall bug recover results (✓: recovered; $✓_c$: conditionally recovered; recovering these wrong-output failures requires annotations.)

```
1 //Thread 1                                  1 //Thread 2
2 fprintf("Start %d",Init);                   2 //End is 0 until below
3 tmp=End;                                    3
4 assert(tmp>0);                              4 End=time(NULL);
5 fprintf("Stop %d, Total %d", tmp, tmp-Init);
```

Figure 9. An atomicity/order violation in FFT that causes a wrong-output failure. If developers specify the output-correctness condition (e.g., the assert above), ConAir can help recover the failure.

multi-threaded software to survive hidden bugs; (3) how ConAir achieves negligible run-time overhead; (4) the fast failure recovery under ConAir; (5) the static analysis time of ConAir.

6.1 Failure recovery

6.1.1 Fix-mode failure recovery

In fix mode, ConAir is aware of the failure sites and failure symptoms. It inserts rollback-recovery code accordingly.

Among the non-deadlock bugs that are evaluated, five of them (FFT, HTTrack, MozillaXP, Transmission, and ZSNES) cause failures in a thread that reads a shared variable too early; FFT[6] and MySQL2 cause failures due to RAR atomicity violations; MySQL1 causes failures due to a WAW atomicity violation. ConAir can successfully recover failures caused by all of them.

Some failure recoveries only roll back a few instructions. For example, Figure 9 shows a bug in FFT. In this program, thread 1 could unexpectedly read End (line 3in Figure 9) before thread 2 updates it, causing either an order violation or an atomicity violation and a wrong-output failure. ConAir inserts a setjmp right before the assert, which helps FFT to recover this failure.

Two of these 10 bugs (Transmission and MozillaXP) require inter-procedural reexecution to recover. For example, Figure 10 depicts the MozillaXP bug. In MozillaXP, thread 1 could unexpectedly read mThd->state in function GetState before the global pointer mThd is initialized by thread 2. This could cause a segmentation-fault failure. ConAir inserts a pointer sanity check right before line 9 in GetState; it also identifies a reexecution point inside function Get and inserts setjmp there. Once ConAir sees an invalid pointer at line 9 in thread 1, the program will automatically jump back to before the invocation of GetState in Get. Eventually, thread 2 will initialize mThd and the program will succeed.

Deadlock recovery is slightly different from the recovery of non-deadlock bugs. Figure 11 shows a real-world deadlock bug in HawkNL. As we can see, thread 1 and thread 2 could acquire nlock and slock in reversed orders and lead to a deadlock. ConAir analyzes both threads. When ConAir considers Lock(&slock) (line 8) in

[6] FFT contains both order violations and atomicity violations.

```
1 //Thread 1                                  1 //Thread 2
2 Get(){                                      2 //mThd is shared
3   ...                                       3 //between two threads;
4   tmp=GetState(mThd);                       4 //it is 0 before
5 }                                           5 //initialized below.
6                                             6
7 GetState(THD *thd)                          7 InitThd(){
8 {                                           8
9   return(thd->state &                       9   mThd =
10        THREAD_DETACHED);                   10    CreateThd(..);
11 }                                          11 }
```

Figure 10. An order violation in Mozilla XPCOM.

```
1 //Thread 1                                  1 //Thread 2
2 Close(){                                    2 Shutdown(){
3   ...                                       3   ...
4   Lock(&nlock);                             4   Lock(&slock);
5                                             5   if(nSockets!=NULL){
6   driver->Close();                          6     int i=0;
7                                             7     if(nSockets[i]){
8   Lock(&slock);                             8       Lock(&nlock);
9   ...                                       9       ...
10 }                                          10    }
                                              11  }
                                              12 }
```

Figure 11. A deadlock in HawkNL.

thread 1 as a potential failure site, the reexecution region is very short due to the idempotency-destroying operation, driver->Close(). Since this region does not contain another lock acquisition function, ConAir considers it as unrecoverable and does not attempt any failure recovery in thread 1 (Section 4.2). When ConAir considers Lock(&nlock) (line 8) in thread 2 as a potential failure site, its reexecution region can go all the way back to before the invocation of Lock(&slock) (line 4) in thread 2. Since this region contains another lock-acquisition function, ConAir considers Lock(&nlock) in thread 2 as a recoverable failure site. ConAir turns it into a lock with timeout and inserts setjmp to the beginning of Shutdown function. At run time, once thread 2 times out at its attempt to acquire nlock, thread 2 will release slock and reexecute a large chunk of Shutdown. This effectively resolves the deadlock problem in HawkNL.

Summary ConAir can effectively fix concurrency bugs with a variety of root causes once the failure sites and symptoms are known.

6.1.2 Survival mode

In survival mode, ConAir is **not** aware of any bug. It automatically and systematically identifies potential failure sites and transforms the program accordingly.

As shown in Table 4, ConAir has identified and hardened 7 – 19185 static failure sites in each benchmark program. Naturally, ConAir identifies the fewest failure sites in the smallest programs (FFT and HawkNL) and the most failure sites in the largest programs (MySQL1 and MySQL2). In general, potential segmentation-fault sites dominate all types of potential failure sites, because ConAir identifies every heap/global pointer dereference as a potential segmentation-fault site. Potential deadlock sites are the fewest among all four types of failure sites, because only a lock operation that is enclosed by another lock operation with no write to shared variables in between is identified as a potential deadlock site that is recoverable by ConAir. HTTrack developers left many assertions in the program, leading to a large number of potential assertion-violation sites.

App.	Assertion Violation	Wrong Output	Seg. Fault	Dead-lock	Total
FFT	5	34	14	0	53
HawkNL	0	0	5	2	7
HTTrack	657	504	3146	0	4307
MozillaXP	1	117	6791	0	6909
MozillaJS	0	5	134	6	146
MYSQL1	119	3256	15791	19	19185
MYSQL2	518	2853	15498	21	18890
SQLite	0	25	47	1	73
Transmission	430	190	2151	0	2771
ZSNES	1	50	331	0	382

Table 4. Static failure sites hardened by ConAir

App.	Survival Mode		Fix Mode	
	Static	Dynamic	Static	Dynamic
FFT	56	24	5	5
HawkNL	7	7	1	1
HTTrack	3570	12995	3	4
MozillaXP	3647	2170	1	23
MozillaJS	144	6	1	1
MYSQL1	12494	215218	1	20
MYSQL2	13031	82394	1	30
SQLite	142	7	1	1
Transmission	2568	4425	3	8
ZSNES	321	32	1	2

Table 5. The number of reexecution points inserted by ConAir

These automatically identified potential failure sites include the failure sites of all the 10 bugs that are evaluated. Therefore, ConAir can help software successfully recover from these hidden bugs.

Note that survival-mode ConAir identifies every output functions, including fprintf, printf, application-specific functions, such as my_printf in MySQL and js_printf in Mozilla, and others as a potential site of wrong output. The current prototype of ConAir needs developers' specification to recover a wrong-output failure, as shown in Figure 9. We believe this effort is worthwhile for hardening critical outputs. Future work can also use likely-invariant inference tools [15] to infer such specifications for an output function, and automate the wrong-output failure recovery process.

Summary The above evaluation shows that ConAir is effective to help software survive failures caused by hidden bugs.

6.2 Runtime overhead

The run-time overhead of ConAir comes from four sources: (1) code inserted at every reexecution point; (2) extra condition-checking at the failure sites, such as sanity checking for pointers at potential segmentation-fault sites; (3) code inserted at call site of memory-allocation and lock functions. (4) using the −no−stack−slot−sharing LLVM linking flag. Among these four, the first one is the dominant source.

To understand the runtime overhead of ConAir, we have counted the number of reexecution points in the hardened programs.

As shown in Table 5, ConAir introduces 6 – 215218 dynamic reexecution points in survival mode. Considering that each reexecution point only takes a few nanoseconds to execute (a setjmp and a local counter increment), the low overhead of survival-mode ConAir is understandable. Naturally, the fix-mode ConAir introduces only a few reexecution points, as shown in Table 5. Its overhead is not perceivable.

There are mainly two reasons that ConAir only requires a relatively small numbers of reexecution points. First, the reexecution points are identified according to potential failure sites. Different from previous work [12], ConAir does not aim to find a reexecution point for every instruction in the program. Instead, it targets on common failures of concurrency bugs. Second, ConAir optimization discussed in Section 4.2 has helped to remove failure sites that are not recoverable under ConAir and corresponding reexecution points.

App.	Non-Deadlock		Deadlock	
	Static	Dynamic	Static	Dynamic
FFT	2.0%	5.0%	N/A	N/A
HawkNL	50%	50%	33%	83%
HTTrack	42%	5.4%	N/A	N/A
MozillaXP	2.4%	1.7%	N/A	N/A
MozillaJS	0.0%	0.0%	50%	50%
MYSQL1	1.1%	8.2%	88%	99%
MYSQL2	0.46%	14.6%	91%	100%
SQLite	3.4%	0.0%	30%	71%
Transmission	4.5%	1.76%	N/A	N/A
ZSNES	6.8%	36.4%	N/A	N/A

Table 6. The percentage of reexecution points that are optimized (N/A: the non-optimized version has 0 reexecution point).

Application	ConAir Recovery		Restart
	Time (μs)	# Retries	Time (μs)
FFT	907	97	3189072
HawkNL	59	1	943
HTTrack	4237	474	10776
MozillaXP	17388	8432	207041
MozillaJS	44	1	472
MYSQL1	6014	575	26308
MYSQL2	8	1	836177
SQLite	86	1	1443
Transmission	6476	761	553109
ZSNES	1022	123	8643

Table 7. Failure recovery time (The experiments are conducted with small amount of noises inserted to help trigger the concurrency-bug failures).

To quantitatively demonstrates the optimization effect, we have tried to harden each program by survival-mode ConAir with and without ConAir optimization. As we can see in Table 6, the optimization effect is significant for deadlock reexecution points: 30–91% of static reexecution points can be optimized away. Many lock operations are not enclosed by another lock operation in its reexecution region, and hence are considered as not recoverable. In comparison, the optimization effect for non-deadlock reexecution points is not as significant. Fewer than 10% of static or dynamic reexecution points are optimized away for most benchmarks. The reason is that the optimization cannot eliminate any segmentation-fault reexecution points. In the current prototype of ConAir, the potential site of a segmentation fault is the dereference of a global/heap pointer variable. Since the reexecution regions of this type of failure sites always contain a read of global/heap variable (i.e., the pointer) that can affect the failure outcome, ConAir considers them un-optimizable. HTTrack has a large number of reexecution points that are not related to segmentation faults. Therefore, a significant number of its reexecution points are optimized away.

Summary Benefiting from its single-threaded idempotent reexecution design, its failure-oriented idempotent region identification, and its optimization analysis, ConAir can effectively improve the reliability of production-run software almost for free.

6.3 Recovery time

Recovery time affects the availability of production-run software. We quantitatively measure the failure-recovery time under ConAir, and compare it with the time of restarting the whole program.

Note that software restart almost always changes the program semantics perceived by users, unless it can log all the inputs and external signals, and sandbox I/O operations. In addition, the recovery time of software restart becomes worse with the workload getting larger. Instead, the recovery time of ConAir is largely oblivious of the workload. Therefore, the advantage of ConAir recovery in practice would be much more significant than the quantitative results presented below.

As shown in Table 7, the failure recovery in ConAir ranges between 8 microseconds and 17 milliseconds. In contrast, program restart could take as long as several seconds when the failure occurs at the end of a scientific computation (FFT). The recovery-time speedup of ConAir ranges from 8 times to over 100,000 times.

The ConAir recovery speed is mainly determined by the root cause of the failure. Failures caused by RAR atomicity violations (Figure 2c) are always fast to recover. The failing thread does not need to wait for any other thread. Once it reexecutes the read-after-read, the atomicity violation is immediately eliminated and the failure immediately recovers. That is why MySQL2 takes only 8 microseconds to recover. Deadlock bugs (HawkNL, SQLite and MozillaJS) also require relatively short recovery time. After one thread t_1 involved in the deadlock releases a lock at the failure site, another thread t_2 can almost immediately jumps out of the deadlock situation. The recovery time for t_1 will be determined by the critical region length of t_2. Failures caused by order violations usually require a relatively long time to recover. Take the MozillaXP bug shown in Figure 10 as an example. At run time, thread 1 reads mThd too early and has to rollback due to an invalid value in mThd. Rolling back thread 1 once may not recover the failure, because thread 1 has to wait for thread 2's progress. In our experiment, this rollback is conducted more than 8000 times until thread 2 initializes mThd. This is the main reason of the relatively long recovery time of HTTrack, MozillaXP, Transmission, and ZSNES.

Summary Our evaluation shows that ConAir supports fast failure recovery. It can help software survive failures with little impact to latency and availability.

6.4 Static analysis time

The static analysis and code transformation time of ConAir ranges from less than a second (FFT) to around 4 hours (MySQL). The majority of the time is spent in attempting inter-procedural failure recovery. In fact, the basic intra-procedural static analysis discussed in Section 3 and the optimization analysis discussed in Section 4.2 together take only 50 seconds for MySQL and fewer than 10 seconds for other benchmarks.

Summary The static analysis of ConAir is fast enough to process large real-world multi-threaded software. If the time budget is tight, ConAir users can disable the inter-procedural recovery analysis.

6.5 Limitations of ConAir

ConAir does not aim to recover all concurrency-bug failures, which inevitably requires much higher run-time overhead and/or complicated platform support. Specifically, ConAir cannot recover failures that require multi-threaded reexecution or very long reexecution regions, as discussed in Section 2.1. Fortunately, as also discussed in Section 2.1, many real-world concurrency bugs do not require multi-threaded reexecution or long reexecution to recover, and hence can benefit from ConAir. Finally, ConAir cannot recover software from a wrong-output failure, if developers do not provide output-correctness conditions.

7. Related Work

Many closely related works have been discussed in the earlier sections. This section presents other related works.

Concurrency bug detection Many techniques have been proposed to detect data races [14, 18, 47, 59], atomicity violations [5, 17, 33, 34], order violations [19, 36, 49, 57, 60], and others. Bug-detection tools help developers discover and understand the defects in software. ConAir has a different goal from bug-detection tools. It aims to recover concurrency-bug failures at run time without understanding the bug root causes.

Software checkpoint and replay Checkpoint and replay are useful techniques for failure diagnosis and recovery. Many techniques have been proposed to checkpoint and replay multi-threaded software deterministically or non-deterministically [1, 22, 26, 28]. To achieve good performance, these techniques often require sophisticated operating-system support or hardware support. ConAir only rolls back an idempotent region in one thread and does not require these sophisticated techniques.

Deterministic execution Deterministic systems [2, 3, 7, 32, 41] force a multi-threaded program to execute a deterministic interleaving under a given input. This promising approach still faces challenges, such as overhead, integration with system non-determinism, language design, etc. In general, these tools address different problems from ConAir. Even inside a deterministic run time, concurrency bugs can still occur and require recovery.

Rollback recovery As discussed in Section 1, several rollback-recovery systems have been built before, such as Rx [44], ASSURE [50], and Frost [53]. They all change operating systems to support whole program checkpoint and rollback. Rx changes the program environment during reexecution to handle deterministic bugs. ASSURE rolls back a failed software to an existing error-handling path. It is designed to mitigate the impact of deterministic bugs, and cannot help software generate correct results after the manifestation of a non-deterministic concurrency bug. Frost [53] proposes a novel solution to survive data races. With OS support, it executes multiple replicas of the program with complementary thread schedules at the same time. Periodically, it compares the states of different replicas and tries to survive state divergence caused by data races. In general, these systems all require checkpointing the whole program states and rolling back all threads during a failure. Consequently, they all require sophisticated changes to operating systems.

Microreboot [4] is a recovery technique that reboots only application components, instead of the whole program, when failures occur. To benefit from microreboot, the programmers have to manually separate their systems into components (groups of objects) that can be individually restarted, such as Enterprise Java Beans components in J2EE applications. ConAir shares a common high-level philosophy with microreboot of not rolling back the whole program. However, the similarity ends there. ConAir focuses on concurrency-bug failure recovery. It works on any C/C++ multi-threaded software without manual changes. It automatically identifies reexecution points and conducts automated code transformation.

Apart from rollback recovery, a recent work studies the phenomenon that some software is able to automatically recover from state corruption, because they overwrite the corrupted states with new input data. This type of software is called self-stabilizing programs [13]. To some extent, ConAir can transform a multi-threaded program to become self-stabilizing.

Idempotency While the idea of leveraging idempotency for recovery is not new [9–12, 16, 21, 25, 38], our work is the first to apply it towards the problem of recovery from concurrency bugs. Additionally, most previous work on idempotency has assumed hardware support for recovery with a focus on hardware exceptions [9, 21, 38], hardware faults [11, 16], and hardware mis-speculation [25]. Our

technique requires no hardware support. While the general paradigm of idempotent processing [12], which allows programs to be executed entirely as sequences of idempotent regions, does not strictly require hardware support to enable various features, the authors' technique does *not* work for general multi-threaded programs. This technique allows an idempotent region to store to shared variables. Such a region cannot be considered idempotent in the presence of data races and hence their algorithm cannot be used. In addition, instead of splitting the entire program into idempotent regions, ConAir only identifies idempotent regions that end at potential concurrency-bug failure sites. This focused approach allows ConAir to achieve negligible overhead ($<$1%), while previous work could incur more than 10% overhead [12]. We furthermore do not limit ourselves to intra-procedural analysis and allow our static analysis to be applied inter-procedurally to maximize its effectiveness.

8. Conclusions

This paper presents ConAir, a static analysis and code transformation tool that helps fix and survive concurrency-bug failures through single-threaded recovery and idempotent processing. The evaluation using 10 real-world concurrency bugs shows that ConAir successfully helps software quickly recover from failures that cover a variety of symptoms and root causes. ConAir works well even when it has no knowledge about a bug.

ConAir is not designed to recover all failures, but is effective for a large number of common concurrency-bug failures. It only introduces negligible run-time overhead, less than 1% in our experiments. This good performance is achieved without any change to hardware or operating systems and is suitable for production-run deployment. ConAir's effectiveness is a result of a seemingly serendipitous property: short recovery regions are naturally idempotent. In future work, we hope to investigate whether automatic or programmer-aided transformations can help increase its coverage.

ConAir provides a novel use of existing assertions and error checking in programs. With ConAir, assertions and error-checking code no longer just passively observe system failures and errors. Instead, they actively help ConAir to recover software from failures and correct software internal errors. ConAir's creative use of assertions opens up the possibility of sanity checks in multi-threaded programs being useful in deployment and as a recovery tool beyond just a debugging tool. For the future work, we would like to investigate how well this works in the field on widely deployed and used code-bases. Also, we would like to understand developer issues in using such a paradigm.

ConAir introduces a perspective that many points in the design space of rollback/recovery are meaningful, with reexecution regions spanning from tens of instructions to the whole program. Future work can extend ConAir to explore other design points in this large design space.

Acknowledgments

We thank the anonymous reviewers for their invaluable feedback. We thank Prof. Mark Hill for his invaluable suggestions. We thank John T. Criswell for going above and beyond his responsibilities to answer our LLVM related questions. This work is supported by NSF grants CCF-0845751, CCF-1018180, CCF-1054616, CCF-1217582, a Google U.S./Canada PhD fellowship, and a Claire Boothe Luce faculty fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

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

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, 2004.

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

[6] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, 2008.

[7] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Trans. Program. Lang. Syst.*, 13(4), Oct. 1991.

[9] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *MICRO*, 2011.

[10] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *CGO*, 2013.

[11] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[12] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *PLDI*, 2012.

[13] Y. h. Eom and B. Demsky. Self-stabilizing java. In *PLDI*, 2012.

[14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

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

[16] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *MICRO*, 2011.

[17] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

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

[19] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[20] P. Godefroid and N. Nagappani. Concurrency at Microsoft – an exploratory survey. Technical report, MSR-TR-2008-75, Microsoft Research, May 2008.

[21] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *ICS*, 2006.

[22] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Commun. ACM*, 52(6), June 2009.

[23] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

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

[25] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst.*, 28(5), Sept. 2006.

[26] S. T. King, G. W. Dunlap, and P. M. Chen. Operating systems with time-traveling virtual machines. In *Usenix*, 2005.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[28] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS*, 2010.

[29] Z. Letko, T. Vojnar, and B. Křena. AtomRace: data race and atomicity violation detector and healer. In *PADTAD*, 2008.

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

[31] Z. Li, L. Tan, X. Wang, Y. Zhou, and C. Zhai. An empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[32] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

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

[34] 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*, 2007.

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

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

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

[38] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4), Nov. 1993.

[39] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. MIT-CSAIL-TR-2010-038.

[40] MySQL. Mysql 5.6 reference manual. http://dev.mysql.com/doc/refman/5.6/en/.

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

[42] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/businesscenter/article/255911/ nasdaqs_facebook_glitch_came_from_race_conditions.html.

[43] S. Qi, N. Otsuki, L. O. Nogueira, A. Muzahid, and J. Torrellas. Pacman: Tolerating asymmetric data races with unintrusive hardware. In *HPCA*, 2012.

[44] 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.

[45] S. K. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS*, 2009.

[46] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. G. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPOPP*, 2009.

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

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

[49] 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.

[50] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, 2009.

[51] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Automatic atomic region identification in shared memory SPMD programs. In *OOPSLA*, 2010.

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

[53] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, 2011.

[54] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[55] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[56] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

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

[58] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, 2010.

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

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

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