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

Wei Zhang  Marc de Kruijf  Ang Li
Shan Lu  Karthikeyan Sankaralingam

ASPLOS 2013
Overview

- Motivation
- ConAir System
- Evaluation
- Criticism
Motivation - Overview

- Concurrency bugs remain hidden
- Severe failure
- Hard to fix
Motivation – Automatic Tools

- Compatibility
- Correctness
- Generality
- Performance
Motivation – Automatic Fixing

- Additional Synchronisation
- Needs to know about the root cause
Motivation – Prohibit Interleaving

- Additional serialisation leads to performance loss
- Only works on certain interleavings
- Some approaches need programmer annotations
Motivation – Traditional Rollback Recovery

- Require checkpointing
- All threads are rolled back
- Require OS/hardware modification to run efficiently
ConAir - Overview

- Single thread rollback
- No checkpoints
- Static program modification
ConAir - Feasibility

- Atomicity violations
  - About 70% of non-deadlock bugs
- Rollback of one thread establishes serialised execution
- About 92% recoverable
ConAir – Feasibility

- Write after write (WAW)

  Log = CLOSE;
  Log = OPEN;
  If (Log != OPEN)
  { //output failure}
ConAir – Feasibility

- Write after write (WAW)

Log = CLOSE;
Log = OPEN;

If (Log != OPEN)
{ //output failure}

Rollback of this thread recovers the program
ConAir - Feasibility

- Read after write (RAW)

```
ptr = a.ptr;
tmp = *ptr;
ptr = NULL;
```

ConAir - Feasibility

- Read after write (RAW)

```c
ptr = aptr;
tmp = *ptr;
ptr = NULL;
```

Rollback of this thread recovers the program
ConAir - Feasibility

- Read after read (RAR)

```c
if (ptr)
{ fputs(ptr); }
ptr = NULL;
```
ConAir - Feasibility

- Read after read (RAR)

```c
if (ptr)
{
  fputs(ptr);
}
ptr = NULL;
```

Rollback of this thread recovers
ConAir - Feasibility

- Write after read (WAR)

```c
count += deposit1;
cnt += deposit2;
printf(cnt);
```
ConAir - Feasibility

- Write after read (WAR)

```c
count += deposit1;          cnt += deposit2;
printf(cnt);
```

Rollback of this thread recovers
ConAir - Feasibility

- Order-violation
  - 30% of non-deadlock bugs
- About 50% recoverable

```java
SomeClass b = NULL; b.foo();
b = new SomeClass();
```
ConAir - Feasibility

- **Order-violation**
  - 30% of non-deadlock bugs
- About 50% recoverable

```java
SomeClass b = NULL;
b = new SomeClass();
b.foo();
```

Rollback of this thread recovers
ConAir - Feasibility

- Deadlock bugs
  - About 40% of bugs
- Recovers if one thread rolled back
ConAir - Basics

- Identify potential failure sites
- Identify idempotent region for each failure site
- Insert recovery code
- Two modes: Fix and Survival
ConAir – Failure Site Identification

• Survival mode
  - Assertions
    • Can use assertion to indicate output failure
  - Heap/global pointer dereference
  - Deadlock detection with any detection tool
ConAir – Failure Site Identification

- Fix mode
  - Programmer specifies the location of the failure
ConAir – Idempotent Regions

- Re-execution only on idempotent regions
- Guarantees correctness
- May be too weak for many bugs
  - But has a low overhead
ConAir – Idempotent Regions

- No writes to shared Variables
- No I/O
- No idempotent destroying write to local variables

```c
x = x+1;
y = x+1;
z = x+y;
z = x+y;
```
ConAir – Idempotent Regions

- No writes to shared Variables
- No I/O
- No idempotent destroying write to local variables

\[
\begin{align*}
x &= x+1; & y &= x+1; \\
z &= x+y; & z &= x+y;
\end{align*}
\]

Not idempotent, value of x changes in each re-execution
ConAir – Idempotent Regions

- Discovery non-trivial
  - Source code ↔ bit code ↔ binary code
- Binary code analysis alone complicated
- Search all backward paths from failure sites
ConAir – Idempotent Regions

- Weaken the definition
  - Idempotent function calls
  - Parent functions
- Requires more analysis
ConAir – Recovery Code

- At start of re-execution region: setjmp
  - Saves the register image
- At failure site: longjmp
  - Loads the register image and executes from setjmp
- Multiple retries
ConAir – Recovery Code

- Original code

```c
...  
if(e){
    ...
} else {

    __assert_fail(...);

}
```
ConAir – Recovery Code

• Modified code

```
__thread jmp_buf c;
__thread int RetryCnt=0;
...
Reexecution:
    setjmp(c);
    ... //idempotent region
    if(e){
        ...
    } else {
        Failure:
            while(RetryCnt++<maxRetryNum){
                longjmp(c, 0);
            }
            __assert_fail(...);
    }
```
ConAir – Recovery Code

- Modified code

```c
__thread jmp_buf c;
__thread int RetryCnt=0;
...
Reexecution:
  setjmp(c);
  ... //idempotent region
  if(e){
    ...
  } else {
    Failure:
      while(RetryCnt++<maxRetryNum){
        longjmp(c, 0);
      }
      __assert_fail(...);
  }
```
ConAir – Recovery Code

- **Modified code**

```c
__thread jmp_buf c;
__thread int RetryCnt=0;
...

Reexecution:
setjmp(c);
... //idempotent region
if(e){
... }
} else {

Failure:
while(RetryCnt++<maxRetryNum){
    longjmp(c, 0);
} __assert_fail(...);
}
```
ConAir – Recovery Code

- **Modified code**

```c
__thread jmp_buf c;
__thread int RetryCnt=0;
...
Reexecution:
  setjmp(c);
  ...
  //idempotent region
  if(e){
  ...
  } else {
Failure:
  while(RetryCnt++<maxRetryNum){
    longjmp(c, 0);
  }
  __assert_fail(...);
}
```

- Thread local variables for registers and retry count
- Registers saved and jump point set
- Try multiple times, insert random sleep for deadlock recovery
ConAir – Recovery Code

- Modified code

```c
__thread jmp_buf c;
__thread int RetryCnt=0;
...

Reexecution:
setjmp(c);
... //idempotent region
if(e){
...
} else {
Failure:
while(RetryCnt++<maxRetryNum){
    longjmp(c, 0);
}
    __assert_fail(...);
}
```

Thread local variables for registers and retry count

Registers saved and jump point set

Try multiple times, insert random sleep for deadlock recovery

Jump to the saved location and restore registers
ConAir - Optimisations

- Allow library functions
  - Extend idempotent region
  - Needed for certain recoveries (Deadlock)
  - Need compensation function (lock/unlock, malloc/free)
- ConAir allows malloc and lock in idempotent regions
  - Cleanup before longjmp
ConAir – Optimisations

- Remove code from unrecoverable fail sites
  - Statically proven
  - Deadlock recovery with no lock in idempotent region
  - Non-deadlock recovery with no shared-variable reads
ConAir - Optimisations

- Include parent functions in idempotent region
- Should at least change one argument
- Significant overhead in static analysis
Evaluation

- 10 bugs in open-source libraries
- Wide variety of root causes and failure symptoms
- Analyze performance, overhead and recovery time
- Also analyze static analysis time
Evaluation

- Modify buggy code with sleep instructions
  - Almost 100% failure rate
- Run 1000 times with applied ConAir
- Successfully recovered if none causes the bug
Evaluation

- Run time overhead measured on the original source code
Results – Fix Mode

- No measurable overhead
  - Small number of failure sites
- Recovered all failures
Results – Survival Mode

- Small overhead (<1%)
- Could recover 8/10 bugs
  - I/O operations would require program annotations
Results – Recovery Time

- At most 17 milliseconds
- Much better than crash/program restart
Results – Static Analysis

- FFT (1.2K lines of code)
  - Less than a second
- MySQL (~685k lines of code)
  - 4 hours
- Inter procedural analysis requires big part
  - Only 50 seconds in MySQL are spent on intra procedural analysis
Limitations

- No completeness
Criticism

- Aims of ConAir are met
- Surprisingly low overhead
- Easy to read, enough explanations
- Hard to find negative points, since ConAir is more of a heuristic approach