Multithreaded Software is everywhere
Multithreaded software is difficult to write

Need to think about many threads instead of one
Threads interact via shared memory

Reasoning about concurrent shared accesses is hard
Behavior can change from one execution to the next
Some behavior is bad, like a crash or hang

Key problem: understanding why bad things happen
Typically debug with a core dump

This thread crashed @ “assert(ptr != NULL)”

ptr = NULL
Typically debug with a core dump

Core dump tells us what happened, not why.

ptr = NULL

Why!?!?

This thread crashed @ "assert(ptr != NULL)"
Our work answers the **why?** question.

**Last Writer Slices** record each value’s provenance.

- This thread set $x = \text{NULL}$ right here.
- This thread crashed @ “assert(ptr != NULL)”

$$\text{ptr} = \text{NULL}$$
Bonus: provenance reveals communication

LWS
This thread wrote x here...

CTraps
Check LWS... Communication! Reader != last writer

Communication Traps: custom communication handlers

CT_Handler(...){
  build_c_graph();
  check_atomicity();
  coop_bug_iso();
}
CTraps
Executes application-specific handlers when threads communicate

Memory
Tracks data provenance at runtime with low overhead

Multi-threaded Execution

LWS
Informs of writes
Informs of communication

Debugging
Programmer examines provenance via LWS

Analysis
Arbitrary concurrency analyses via CTraps

Efficiency
Overheads low enough for production use
append()

len = len + 1
realloc(str, len)
str[len - 1] = 'a'

**Shared Variables**

- len: length of string
- str: string buffer
len: length of string
str: string buffer

**Shared Variables**

len = len + 1

append()

str[len-1] = 'a'

realloc(str, len)

len = 0, str = []

len = 1, str = []

len = 1, str = [\_]

len = 1, str = [a]
len = len + 1

realloc(str, len)

str[len-1] = 'a'

\[\text{Crash: str}[len-1] \text{ out of bounds}\]
Shared Variables

len: length of string
str: string buffer

Programmer: “This must be wrong”

len = len + 1
realloc(str, len)
str[len-1] = ‘a’

Crash: str[len-1] out of bounds
Data Provenance Tracking for Concurrent Programs - Brandon Lucia

Programmer: “One of these must be wrong”

Shared Variables

len: length of string
str: string buffer

append()

len = len + 1
 realloc(str, len)
 str[len-1] = ‘a’

append()

len = len + 1
 realloc(str, len)
 str[len-1] = ‘a’

erase()

len = len - 1
Programmer: “WTF?!"
Shared Variables

len: length of string
str: string buffer

len = len + 1
realloc(str, len)
str[len - 1] = 'a'

Last Writer Slices tracks data provenance: thread & code point that last wrote len
T1
append()

len = len+1

realloc(str,len)

str[len-1] = ‘a’

T2
append()

len = len+1

realloc(str,len)

str[len-1] = ‘a’
T1
append()

len = len + 1

realloc(str, len)

str[len-1] = 'a'

T2
append()

len = len + 1

realloc(str, len)

str[len-1] = 'a'

Last Writer Table

<table>
<thead>
<tr>
<th>len</th>
<th>T1</th>
<th>B</th>
</tr>
</thead>
</table>

Read Operation

Write Operation
T1
append()

len = len+1

realloc(str,len)

str[len-1] = 'a'

T2
append()

len = len+1

realloc(str,len)

str[len-1] = 'a'

Read Operation

Write Operation

Last Writer Table

len T2 B

Update
len = len + 1
append()
realloc(str, len)
str[len-1] = 'a'

T1
append()

len = len + 1
append()
realloc(str, len)
str[len-1] = 'a'

T2
append()

Read Operation

Write Operation

Last Writer Table

len T2 B

Crash

Data Provenance Tracking for Concurrent Programs - Brandon Lucia
len = len+1
append()
realloc(str,len)
str[len-1] = 'a'

T1

T2

B
len = len+1
append()
realloc(str,len)
str[len-1] = 'a'

E

Breakpoint

Read Operation

Write Operation

Last Writer Table
T1
append()

\[ \text{len} = \text{len} + 1 \]

realloc(str, \text{len})

str[\text{len}-1] = 'a'

Reads are free for LWS

T2
append()

\[ \text{len} = \text{len} + 1 \]

realloc(str, \text{len})

str[\text{len}-1] = 'a'
CTraps Key Idea:
Different thread in the LWT? Threads are communicating.
CTraps allows **communication handlers**

```c
append()
len = len+1
realloc(str,len)
str[len-1] = 'a'
```

**CT_Handler**:

```c
CT_Handler(current_code_pt,
current_thread,
LWS_code_pt,
LWS_thread,
mem_addr){
    add_comm_graph_edge(current_code_pt,
    [Lucia, MICRO '09; PLDI '11;
    LWS_code_pt);
    Shi, OOPSLA '10;
    Gao, SC '07;]
}
```

Handlers implement arbitrary **communication analysis**
CTraps allows communication handlers

```plaintext
len = len+1
append()
realloc(str,len)
str[len-1] = 'a'
```

Handlers implement arbitrary communication analysis
Last Writer Slicing & CTraps Implementation

<table>
<thead>
<tr>
<th>len</th>
<th>T2</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>T17</td>
<td>A</td>
</tr>
<tr>
<td>foo</td>
<td>T9</td>
<td>C</td>
</tr>
<tr>
<td>bar</td>
<td>T6</td>
<td>D</td>
</tr>
</tbody>
</table>

Exposing communication to CTraps

- Link Program to Runtimes
- @ Write: update LWT; call CTraps Handler
- @ Read: call CTraps Handler

CTraps Runtime Library
- Maintain List of CT_Handlers
- Call CT_handlers on comm.
- Send LWS + Core Dump to debugger (GDB)
Caveat: LWS Implementation & Data-races

- Update_LWT(len)
  - len = len+1
- realloc(str,len)
- Update_LWT(str)
  - str[len-1] = 'a'

LWT adds no synchronization

- Update_LWT(len)
  - len = len+1

Program synchronization keeps LWT consistent

- Lock()
- Update_LWT(len)
  - len = len+1
- Release()

Correct for DRF programs
  (may be incorrect for racy programs)

ordered!
Evaluating LWS and CTraps
Debugging with LWS
Debugging with LWS
Comparison Point: **Bad Value Origin Tracking**

[Bond, et al OOPSLA '07]

```c
ptr = NULL;

len
x

Cleverly implemented using value ‘piggybacking’

if (x == 100){
  ...
}

Check

“A Undefined value originating at used in conditional”

foo()
LWS tracks all values, BVOT only unusable ones
Failing Execution

init() 
lock = new lock() 
A
acquire(lock); ! Crash: lock not initialized 
lock | 0 0

update() 

Non-Failing Execution

init() 
lock = new lock() 
A
acquire(lock); OK: lock initialized 
lock | T1 A

update()
Failing Execution

init()

update()

lock = new lock()

acquire(lock):

Crash: lock not initialized

Non-Failing Execution

init()

update()

lock = new lock()

acquire(lock):

OK: lock initialized

Breakpoint
CTraps Supports Useful Analyses

Communication Graph Collection
[Lucia, MICRO ’09; PLDI ’11;
Shi, OOPSLA ’10; Gao, SC ’07;]

CCI-Prev
[Jin, OOPSLA ’10]

~50 LoC for handlers
~10 LoC for handlers
LWS has overhead low enough for production use
CTraps has practical overhead that scales with analysis complexity.
Systems should track data provenance information

LWS helps with Debugging

CTRaps enables useful Analysis

LWS & CTraps have Efficiency sufficient for production
https://github.com/blucia0a/CTraps-gcc

https://gcc.gnu.org/wiki/plugins

LWS helps with Debugging

CTraps enables useful Analysis

LWS & CTraps have Efficiency sufficient for production
Data Provenance Tracking for Concurrent Programs

Brandon Lucia | Carnegie Mellon University, Dept. of ECE

work done in cooperation with Luis Ceze @ University of Washington, Dept. of CSE