Expositor: scriptable time-travel debugging with first-class traces

Khoo Yit Phang, Jeffrey S. Foster, Michael Hicks
University of Maryland
Debugging

“...we talk a lot about finding bugs, but really, [Firefox’s] bottleneck is not finding bugs but fixing [them]...” – Robert O’Callahan

- Debugging = scientific method in action
  - Programmer makes observations
  - Proposes a hypothesis about the cause of the failure
  - Uses hypothesis to make predictions
  - Tests predictions with experiments
  - Victory? Then fix bug. Else repeat.
Tools for understanding a bug

“[In debugging,] understanding how the failure came to be...requires by far the most time and other resources” – Andreas Zeller

- Many tools/techniques in use
  - Interactive debuggers (e.g., gdb)
  - Print statements, assertions
  - Profilers, visualizers, etc.
  - Record-replay executions (a.k.a., time travel)
    - VMware ReTrace, UndoDB, rudimentary support in gdb, OCaml debugger, …
Scriptable debugging

• Make observations, test hypotheses etc. by writing programs over program executions
  • Benefit: automate tedious repetition
  • Hopefully also:
    • reuse scripts on different programs (or parts of a program)
    • compose old scripts into new ones
    • build sophisticated tools (e.g., visualizations)

• Problem: scripts tend to be brittle, hard to reuse
Counting *foo* calls in GDB

```python
foo = gdb.Breakpoint("foo"); count = 0; more = True
def stop_handler(evt):
    if isinstance(evt, gdb.BreakpointEvent) and foo in evt.breakpoints:
        global count; count += 1
def exit_handler(evt):
    global more; more = False
gdb.events.stop.connect(stop_handler)
gdb.events.exited.connect(exit_handler)
gdb.execute("start")
while more:
    gdb.execute("continue")
gdb.events.exited.disconnect(exit_handler)
gdb.events.stop.disconnect(stop_handler)
foo.delete()
```

- **Set breakpoint at *foo***
- **Function to count when stopped at *foo***
- **Function to detect when program exited***
- **Register handler functions with GDB***
- **Run the program to *main***
- **Keep running until program exited***
- **Deregister handler functions from GDB***
- **Delete the breakpoint**
Classic callback-style programming

- Scripts hard to understand, compose, reuse
  - Control flow is not “straight-line” but smeared across handlers in possibly many different scripts
- Effects conflict
  - One script implemented by setting/disabling breakpoints on particular calls may conflict with a composed script that attempts to count all calls
  - Name clashes on global variables, event names
Expositor

- *Execution trace* is a first-class object:
  - time-indexed list of program state snapshots

- Scripts use common list operations:
  - *filter, map, scan* (like *fold*), *merge* (like *zip*)

- For efficiency, we build on top of time-travel debugging, and use laziness extensively:
  - materialize events only when demanded

- Implemented in Python under GDB and UndoDB
Counting *foo* calls in Expositor

- Create a trace of calls to *foo*:
  
  ```python
  foo = the_execution.breakpoints("foo")
  ```

- Count calls to *foo*:
  
  ```python
  count = len(foo)
  ```

- Control-flow is “straight-line” – list processing

- Easy to reuse and compose scripts:
  
  ```python
  foo_x = foo.filter(lambda s: s.read_var("x") == 0)
  count_x = len(foo_x)
  ```
Expositor API: executions

- **Queries (forces computation):**
  - `the_execution.get_at(t)` – snapshot at time $t$

- **Trace constructors (lazy):**
  - `the_execution.breakpoints/syscalls(name)` – function/sySCALL $name$
  - `the_execution.watchpoints(name, rwflag)` – reads/writes of $name$
  - `the_execution.all_calls/all_returns()` – all function calls/returns

- **Interactive control (forces computation):**
  - `the_execution.cont()` – continue interactive execution
  - `the_execution.get_time()` – get time of paused execution
Expositor API: traces

• **Queries (forces computation):**
  • `trace.__len__()` – number of items, called by `len(trace)`
  • `trace.get_at(t)/get_after(t)/get_before(t)` – item at/after/before time `t`

• **Simple operations (lazy):**
  • `trace.filter(p)` – new trace of items for which `p` returns true
  • `trace.map(f)` – new trace with `f` applied to all items
  • `trace.slice(t0, t1)` – new trace of items from time `t0` to `t1`

• **No add/remove**
  • traces contents completely defined at construction
Trace data structure

```python
foo = the_execution.breakpoints("foo")
```

Dotted boxes are lazy

```python
snapshot = foo.get_before(100)
```

Nodes are annotated with time intervals

This query traverses a lazy node for the first time, so compute the contents

Lazy node since search ended

UndoDB found `foo` at time 50

Search UndoDB starting at time 100

Lazy node since this part is not needed to answer the query
Expositor API: traces

• Complex operations (lazy):
  • $trace_1.merge(f, trace_2)$ – merge $trace_1$ with $trace_2$, using $f$ to merge coincident events
  • $trace_1.[rev_]trailing_merge(f, trace_2)$ – use $f$ to merge events from $trace_1$ with the immediately [succeeding/]preceding elements from $trace_2$
  • $trace.[rev_]scan(f, acc)$ – like fold over [suffixes/]prefixes of $trace$, given an initial accumulator $acc$
  • $trace.[rev_]tscan(f)$ – lazier, associative variant of scan
Merging between traces

tr0
tr1
tr0.merge(f, tr1)

tr0
tr1
None
tr0.trailing_merge(f, tr1)

tr0
tr1
tr0.rev_trailing_merge(f, tr1)
Computing within traces

**scan** is fold over prefixes

\[
\text{tr.\text{scan}(f, acc)}
\]

\[
\text{out}_n = f(\text{out}_{n-1}, \text{in}_n) \mid n > 0
\]

\[
\text{out}_0 = f(\text{acc}, \text{in}_0)
\]

**tscan**: tree-scan using associative \( f \) (written infix)

\[
\text{tr.\text{tscan}(f)}
\]

\[
\text{out}_n = \text{in}_n \ f \ \text{in}_{n-1} \ f \ \ldots \ f \ \text{in}_0
\]

\[
\text{tr.\text{rev}\_\text{tscan}(f)}
\]

\[
\text{out}_n = \text{in}_n \ f \ \text{in}_{n+1} \ f \ \ldots \ f \ \text{in}_{\text{end}}
\]
Laziness in trace operations

- Count calls to $foo(x)$ from end to last $foo(0)$:

  ```python
def count_nonzero_foo(lazy_acc, snapshot):
    if snapshot.read_var("x") != 0:
      return lazy_acc.force() + 1
    else:
      return 0
  
  foo = the_execution.breakpoints("foo")
  nonzero_foo = foo.scan(count_nonzero_foo, 0)
```

For every succeeding item in $foo$ with initial accumulator
Laziness in trace operations

• Count calls to \texttt{foo}(x) from end to last \texttt{foo}(0):

  ```python
  def count_nonzero_foo(lazy_acc, snapshot):
      if snapshot.read_var("x") != 0:
          return lazy_acc.force() + 1
      else:
          return 0
  
  foo = the_execution.breakpoints("foo")
  nonzero_foo = foo.scan(count_nonzero_foo, 0)
  ```

The accumulator is lazy! Why?

By definition of scan: \( acc_n = f (acc_{n-1}, in_n) \)
If the \( acc_{n-1} \) were not lazy, then every input must be computed, compromising laziness
Example: finding stack corruption

calls = the_execution.all_calls()
rets = the_execution.all_returns()
calls_rets = calls.merge(None, rets)
shadow_stacks = calls_rets.map(lambda s: s.retaddrs())

def find_corrupted(snap, opt_shadow):
    if opt_shadow.force() is not None:
        for x, y in zip(snap.read_retaddrs(), opt_shadow.force()):
            if x != y:
                return x
    return None

corrupted_addrs = calls_rets.trailing_merge(find_corrupted, shadow_stacks)
    .filter(lambda x: x is not None)
Example: finding stack corruption, cont’d

% expositor tinyhttpd ➔ Launch Expositor on tinyhttpd
(expositor) python-interactive ➔ Enter GDB’s Python interactive prompt

>>> the_execution.cont()

httpd running on port 47055

Now I pwn your computer

^C

Start tinyhttpd and deploy a stack-smashing exploit

>>> corrupted_addrs = stack_corruption()

>>> t = the_execution.get_time()

>>> last_corrupt = corrupted_addrs.get_before(t)

>>> bad_writes = the_execution.watchpoints(last_corrupt.value, WRITE)

Find the first corrupted address before the execution was paused

>>> last_bad_write = bad_writes.get_before(last_corrupt.time)

Find the write that corrupted the address

% ./exploit.py 47055
Trying port 47055
pwning...
Lazy sets/maps for Expositor

• Problem: laziness compromised when computing with non-lazy sets/maps

```python
def collect_foo_args(lazy_acc, snapshot):
    return lazy_acc.force() \ 
    .union(set([ snapshot.read_var("x") ]))
```

```python
foo = the_execution.breakpoints("foo")
foo_args = foo.scan(collect_foo_args, set())
print (42 in foo_args.get_before(t).value)
```

Forcing `lazy_acc` causes all prior sets to be computed

Must compute all sets before t even if 42 was added at t

Also, each output must be a different set, but `union` does so by deep copying!
EditHAMT: Edit Hash Array Mapped Trie

- Lazy, immutable set/map (multiset/multimap)
  - Each update shares structure with prior EditHAMT, without forcing it to be fully computed
  - Key lookup forces necessary computation

- Combination of:
  - EditList – lazy linked-list of set/map edit operations
  - lazy variant of hash array mapped trie (HAMT) – hashtable/tree hybrid set/map
EditList

- Lazy linked-list based set/map with removal
- lazily append nodes tagged with “add”/“remove”

Dotted lines indicate lazy tail

Space efficient: can be viewed as a succession of sets
HAMT

- Hashtable/tree hybrid
- Chained hashtable ↔ HAMT
  - key hash lookup:
    - bucket array ↔ *array mapped trie* (AMT) – trie-based map from fixed-width integer keys to values
  - hash collision resolution:
    - buckets (e.g., linked-list) ↔ nested HAMTs with different hash functions
AMT

• Trie-based map from integer keys to values

AMT lookup, e.g., \textit{find}(7)
  • \(7 = 00\ 01\ 11\) binary

Lookup are \(O(1)\) since keys are fixed-size

Internal nodes are sparse arrays
Leaf nodes are bindings

\textcolor{red}{Found!}
LazyAMT

• Lazy variant of AMT

Nodes lazily created during lookup

Internal nodes are lazy sparse arrays

Add $57:d$ by creating a new LazyAMT (then forcing the thunks by looking it up)
EditList + hash + LazyAMT = EditHAMT

• Chained hashtable ↔ HAMT ↔ EditHAMT
  • key hash lookup:
    • bucket array ↔ AMT ↔ LazyAMT
  • hash collision resolution:
    • buckets ↔ nested HAMTs ↔ EditLists (also for remove operation)
EditHAMT example

Dotted lines indicate lazy updates

Lookup are \( O(1) \) assuming no hash collisions

Space efficient: can be viewed as a succession of sets
EditHAMT example

EditHAMT at $t_0$ remains unforced

Force prior EditHAMT and share subtree

None $\xrightarrow{\text{"add", } a_0}$ None

Removed, i.e., not found!

• Lookup $t_2$ for $a_0$ $\text{hash}(a_0) = 00\ 01\ 11$ binary
Micro-benchmark: reuse

• Measure advantage of composition and reuse

• Procedure:
  1. Create $trace_1$
  2. Use `get_after/get_before` to enumerate all events from $trace_1$
  3. Repeat for $trace_2$, where $trace_2$ is derived from the same input as $trace_1$
Test debugging target program

- Generic event-generator:

```c
#define LOOP_I 8
#define LOOP_J 8
#define LOOP_K 8
#define LOOP_L 8

void foo(int x, int y) {}
void bar(int z) {}

int main(void) {
    int i, j, k, l;
    for (i = 0; i < LOOP_I; i++) {
        for (j = 0; j < LOOP_J; j++) {
            for (k = 0; k < LOOP_K; k++) bar(k);
            foo(i, j);
            for (l = 0; l < LOOP_L; l++) bar(l);
        }
    }
}
```

- Parameterize number of iterations, relationship between `foo` and `bar`
- Simulate event of interest
- Simulate intervening computation
Two filters over trace

Create `trace1`

Create `trace2`

Query `trace1` get_after

Query `trace2` get_before

```python
foo_trace = the_execution.breakpoints("foo")
trace1 = foo_trace.filter(lambda s: int(s.read_arg("x")) % 2 == 0) # all even x
trace2 = foo_trace.filter(lambda s: int(s.read_arg("x")) % 2 == 1) # all odd x
```
Two filters over trace

```
foo_trace = the_execution.breakpoints("foo")
trace1 = foo_trace.filter(
    lambda s: int(s.read_arg("x")) % 2 == 0)  # all even x
trace2 = foo_trace.filter(
    lambda s: int(s.read_arg("x")) % 2 == 1)  # all odd x
```

**gdb**: using GDB API (forward-only), restart at gray line

**lazy_trace**: Expositor

**strict_trace**: Expositor with trace laziness disabled

**undodb**: using GDB/UndoDB, rewind at gray line

**baseline**: time to simply start the program under GDB/UndoDB

**get_after**: creates/queries traces backwards (except for **gdb**)

**get_before**:
Two filters over trace

```
foo_trace = the_execution.breakpoints("foo")
trace1 = foo_trace.filter(
    lambda s: int(s.read_arg("x")) % 2 == 0)  # all even x
trace2 = foo_trace.filter(
    lambda s: int(s.read_arg("x")) % 2 == 1)  # all odd x
```
Two filters over trace

lazy_trace slope decreases since foo_trace has been computed

strict_trace always slower due to Expositor overhead

period with no matching events

undodb rewind faster than gdb restart

lazy_trace has low cost creation

lazy_trace faster for first 1/3 of queries

get_after

get_before

foo_trace = the_execution.breakpoints("foo")
trace1 = foo_trace.filter(lambda s: int(s.read_arg("x")) % 2 == 0) # all even x
trace2 = foo_trace.filter(lambda s: int(s.read_arg("x")) % 2 == 1) # all odd x
Micro-benchmark: EditHAMT

- Measure advantage of EditHAMT

- Procedure (same input program as before):
  1. Create trace of EditHAMTs mapping $z$ to snapshots of calls to $\text{bar}(z)$
  2. Create trace of events when $\text{foo}(x, y)$ is called containing snapshots of calls to $\text{bar}(z)$ such that $y == z$
    - Computed using $\text{trailing\_merge}$ and the EditHAMT from 1

```
bar(1)   bar(2)   bar(3)
    bar(1)   bar(2)   bar(3)
    foo(1, 2)
        bar(2)
    foo(1, 3)
        bar(3)
```
EditHAMT match *foo to bar*

gdb and lazy_trace_strict_map uses non-lazy
maps (Python dicts) instead of EditHAMTs
EditHAMT match foo to bar

Trace laziness compromised by non-lazy maps

lazy_trace faster than gdb for around the first half of queries

get_after

get_before
Case study: Firefox data race

- Excessive memory usage caused by delayed GC due to data race

Trace of garbage collector behavior

(a) C: gc_call = execution.breakpoints("js_GC")
(b) M: mmap2 = execution.syscalls("mmap2")
(c) timer_trace = set_tracing(A=timer-create, R=timer-fire)
(d) chunkswaiting_trace = execution.watchpoints(gcChunksWaitingToExpire-variable).map(read-gcChunksWaitingToExpire)
(e) chunkswaiting_hb = one_lock(R=gcChunksWaitingToExpire-read, W=gcChunksWaitingToExpire-write, locks, unlocks)

Trace of memory allocation behavior

R: gc_return = execution.breakpoints("js_GC", index=-1)
U: munmap = execution.syscalls("munmap")

Trace of timer events: used an EditHAMT as a shadow set to track timer creation as set addition, timer firing as set removal

Trace of data races: wrote a simple race detector in Expositor

Excessive memory usage caused by delayed GC due to data race
Notable related work

- **Functional reactive programming**
  - MzTake for Racket applies FrTime to debugging Java
  - Here, time always marches forward (eagerly), making it hard to “interact” with an execution

- **Time-travel debuggers**
  - Amber: log every instruction
  - UndoDB, VMware ReTrace, ocamldebug: log only non-determinism + occasional checkpoint snapshots

- **PTQL, PQL: query languages over executions**
  - Implemented as dynamic instrumentation
Summary

• Expositor introduces scripting on execution traces
  • Purely functional
  • Reusable, modular, compositional

• Built on top of time-travel debugging
  • Use laziness for efficiency
  • Could build on top of full captures too
Extra slides
```python
def create_trace(cond, start, cont):
    foo_times = []
    breakpoint = gdb.Breakpoint("foo")
    breakpoint.condition = cond  # "i % 2 == 0" or "i % 2 == 1"
    last_event = [None]
    def stop_handler(event):
        last_event[0] = event
        gdb.events.stop.connect(stop_handler)
    last_snapshot_time = 0
    start()  # gdb.execute("run") to restart, undodb_goto_time(0); gdb.execute("continue") to rewind
                # or undodb_goto_time(0x7fffffffffffffffL); gdb.execute("reverse-continue") to start from the end
    snapshot_time = undodb_get_time()
    while snapshot_time != last_snapshot_time:
        if isinstance(last_event[0], gdb.BreakpointEvent) and breakpoint in last_event[0].breakpoints:
            foo_times.append(snapshot_time)
        elif snapshot_time == last_snapshot_time:
            break
        last_snapshot_time = snapshot_time
    cont()  # gdb.execute("continue") or gdb.execute("reverse-continue")
    snapshot_time = undodb_get_time()
    gdb.events.stop.disconnect(stop_handler)
    breakpoint.delete()
    return foo_times
```

**GDB/UndoDB script**
Filter over filter over trace

get_after

get_before