

# EXPOSITOR: Scriptable Time-Travel Debugging with First-Class Traces

Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks

Computer Science Department, University of Maryland, College Park, MD 20742, USA

{khooyp,jfoster,mwh}@cs.umd.edu

**Abstract**—We present EXPOSITOR, a new debugging environment that combines scripting and time-travel debugging to allow programmers to automate complex debugging tasks. The fundamental abstraction provided by EXPOSITOR is the *execution trace*, which is a time-indexed sequence of program state snapshots or projections thereof. Programmers can manipulate traces as if they were simple lists with operations such as map and filter. Under the hood, EXPOSITOR efficiently implements traces as lazy, sparse interval trees whose contents are materialized on demand. EXPOSITOR also provides a novel data structure, the *edit hash array mapped trie*, which is a lazy implementation of sets, maps, multisets, and multimaps that enables programmers to maximize the efficiency of their debugging scripts. In our micro-benchmarks, EXPOSITOR scripts are faster than the equivalent non-lazy scripts for common debugging scenarios. We have also used EXPOSITOR to debug a stack overflow, and to unravel a subtle data race in Firefox. We believe that EXPOSITOR represents an important step forward in improving the technology for diagnosing complex, hard-to-understand bugs.

**Index Terms**—debugging, time-travel, scripting, lazy data structures, EditHAMT



## 1 INTRODUCTION

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

—Robert O’Callahan [1]

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

—Andreas Zeller [2]

Debugging program failures is an inescapable task for software programmers. Understanding a failure involves repeated application of the scientific method: the programmer makes some observations; proposes a hypothesis as to the cause of the failure; uses this hypothesis to make predictions about the program’s behavior; tests those predictions using experiments; and finally either declares victory or repeats the process with a new or refined hypothesis.

There are certain kinds of bugs that can truly test the mettle of programmers. Large software systems often have complex, subtle, hard-to-understand *mandelbugs*<sup>1</sup> whose untangling can require hours or even days of tedious, hard-to-reuse, seemingly Sisyphean effort. Debugging mandelbugs often requires testing many hypotheses, with lots of backtracking and retries when those hypotheses fail. Standard debuggers make it hard to efficiently reuse the manual effort that goes into hypothesis testing, in particular, it can be hard to juggle the breakpoints, single stepping, and state inspection available in standard debuggers to find the point at which the fault actually happened.

1. “Mandelbug (from the Mandelbrot set): A bug whose underlying causes are so complex and obscure as to make its behavior appear chaotic or even nondeterministic.” From the *New Hacker’s Dictionary* (3d ed.), Raymond E.S., editor, 1996.

*Scriptable debugging* is a powerful technique for hypothesis testing in which programmers write scripts to perform complex debugging tasks. For example, suppose we observe a bug involving a cleverly implemented set data structure. We can try to debug the problem by writing a script that maintains a *shadow data structure* that implements the set more simply (e.g., as a list). We run the buggy program, and the script tracks the program’s calls to insert and remove, stopping execution when the contents of the shadow data structure fail to match those of the buggy one, helping pinpoint the underlying fault.

While we could have employed the same debugging strategy by altering the program itself (e.g., by inserting print statements and assertions), doing so would require recompilation—and that can take considerable time for large programs (e.g., Firefox), thus greatly slowing the rate of hypothesis testing. Modifying a program can also change its behavior—we have all experienced the frustration of inserting a debugging print statement only to make the problem disappear! Scripts also have the benefit that they can invoke libraries not used by the program itself. And, general-purpose scripts may be reused.

### 1.1 Background: Prior Scriptable Debuggers

There has been considerable prior work on scriptable debugging. GDB’s Python interface makes GDB’s interactive commands—stepping, setting breakpoints, etc.—available in a general-purpose programming language. However, this interface employs a callback-oriented programming style which, as pointed out by Marceau et al. [3], reduces composability and reusability as well as complicates checking temporal properties. Marceau et al.

propose treating the program as an event generator—each function call, memory reference, etc. can be thought of as an event—and scripts are written in the style of *functional reactive programming* (FRP) [4]. While FRP-style debugging solves the problems of callback-based programming, it has a key limitation: time always marches forward, so we cannot ask questions about prior states. For example, if while debugging a program we find a doubly freed address, we cannot jump backward in time to find the corresponding malloc. Instead we would need to rerun the program from scratch to find that call, which may be problematic if there is any nondeterminism, e.g., if the addresses returned by malloc differ from run to run. Alternatively, we could prospectively gather the addresses returned by malloc as the program runs, but then we would need to record *all* such calls up to the erroneous free.

Time-travel debuggers, like UndoDB [5], and systems for capturing entire program executions, like Amber [6], allow a single nondeterministic execution to be examined at multiple points in time. Unfortunately, *scriptable* time-travel debuggers typically use callback-style programming, with all its problems. (Sec. 7 discusses prior work in detail.)

## 1.2 EXPOSITOR: Scriptable, Time-Travel Debugging

In this paper, we present EXPOSITOR, a new scriptable debugging system inspired by FRP-style scripting but with the advantages of time-travel debugging. EXPOSITOR scripts treat a program’s *execution trace* as a (potentially infinite) immutable list of time-annotated program state snapshots or projections thereof. Scripts can create or combine traces using common list operations: traces can be filtered, mapped, sliced, folded, and merged to create lightweight projections of the entire program execution. As such, EXPOSITOR is particularly well suited for checking temporal properties of an execution, and for writing new scripts that analyze traces computed by prior scripts. Furthermore, since EXPOSITOR extends GDB’s Python environment and uses the UndoDB [5] time-travel backend for GDB, users can seamlessly switch between running scripts and interacting directly with an execution via GDB. (Sec. 2 overviews EXPOSITOR’s scripting interface.)

The key idea for making EXPOSITOR efficient is to employ *laziness* in its implementation of traces—invoking the time-travel debugger is expensive, and laziness helps minimize the number of calls to it. EXPOSITOR represents traces as sparse, time-indexed interval trees and fills in their contents on demand. For example, suppose we use EXPOSITOR’s breakpoints combinator to create a trace *tr* containing just the program execution’s malloc calls. If we ask for the first element of *tr* before time 42 (perhaps because there is a suspicious program output then), EXPOSITOR will direct the time-travel debugger to time 42 and run it *backward* until hitting the call, capturing the resulting state in the trace data structure. The remainder

of the trace, after time 42 and before the malloc call, is not computed. (Sec. 3 discusses the implementation of traces.)

In addition to traces, EXPOSITOR scripts typically employ various internal data structures to record information, e.g., the set *s* of arguments to malloc calls. These data structures must also be lazy so as not to compromise trace laziness—if we eagerly computed the set *s* just mentioned to answer a membership query at time *t*, we would have to run the time-travel debugger from the start up until *t*, considering all malloc calls, even if only the most recent call is sufficient to satisfy the query. Thus, EXPOSITOR provides script writers with a novel data structure: the *edit hash array mapped trie* (EditHAMT), which provides lazy construction and queries for sets, maps, multisets, and multimaps. As far as we are aware, the EditHAMT is the first data structure to provide these capabilities. (Sec. 4 describes the EditHAMT.)

We have used EXPOSITOR to write a number of simple scripts, as well as to debug two more significant problems. Sec. 2 describes how we used EXPOSITOR to find an exploitable buffer overflow. Sec. 6 explains how we used EXPOSITOR to track down a deep, subtle bug in Firefox that was never directly fixed, though it was papered over with a subsequent bug fix (the fix resolved the symptom, but did not remove the underlying fault). In the process, we developed several reusable analyses, including a simple race detector.

In summary, we believe that EXPOSITOR represents an important step forward in improving the technology for diagnosing complex, hard-to-understand bugs.

## 2 THE DESIGN OF EXPOSITOR

We designed EXPOSITOR to provide programmers with a high-level, declarative API to write analyses over the program execution, as opposed to the low-level, imperative, callback-based API commonly found in other scriptable debuggers.

In particular, the design of EXPOSITOR is based on two key principles. First, the EXPOSITOR API is purely functional—all objects are immutable, and methods manipulate objects by returning new objects. The purely functional API facilitates composition, by reducing the risk of scripts interfering with each other via shared mutable object, as well as reuse, since immutable objects can easily be *memoized* or cached upon construction. It also enables EXPOSITOR to employ lazy programming techniques to improve efficiency.

Second, the trace abstraction provided by EXPOSITOR is based around familiar list-processing APIs found in many languages, such as the built-in list-manipulating functions in Python, the Array methods in JavaScript, the List module in Ocaml, and the Data.List module in Haskell. These APIs are also declarative—programmers manipulate lists using combinators such as filter, map, and merge that operate over entire lists, instead of manipulating individual list elements. These list combinators

```

1 class execution:
2     # get snapshots
3     get_at(t):          snapshot at time t
4
5     # derive traces
6     breakpoints(fn):   snapshot trace of breakpoints at func fn
7     syscalls(fn):     snapshot trace of breakpoints at syscall fn
8     watchpoints(x, rw):
9         snapshot trace of read/write watchpoints at var x
10    all_calls():       snapshot trace of all function entries
11    all_returns():    snapshot trace of all function exits
12
13    # interactive control
14    cont():           manually continue the execution
15    get_time():       latest time of the execution
16
17 class trace:
18     # count/get items
19     __len__():       called by "len(trace)"
20     __iter__():     called by "for item in trace"
21     get_at(t):       item at exactly time t
22     get_after(t):    next item after time t
23     get_before(t):   previous item before time t
24
25     # create a new trace by filtering/mapping a trace
26     filter(p):       subtrace of items for which p returns true
27     map(f):          new trace with f applied to all items
28     slice(t0, t1):   subtrace from time t0 to time t1
29
30     # create a new trace by merging two traces
31     merge(f, tr):    see Fig. 2a
32     trailing_merge(f, tr): see Fig. 2b
33     rev_trailing_merge(f, tr): see Fig. 2c
34
35     # create a new trace by computing over prefixes/suffixes
36     scan(f, acc):    see Fig. 2d
37     rev_scan(f, acc): see Fig. 2e
38     tscan(f, acc):   see Fig. 3a
39     rev_tscan(f, acc): see Fig. 3b
40
41 class item:
42     time:           item's execution time
43     value:          item's contents
44
45 class snapshot:
46     read_var(x):    gdb_value of variable x in current stack frame
47     read_retaddrs():
48         gdb_values of return addresses on the stack
49     backtrace():    print the stack backtrace
50     ... and other methods to access program state ...
51
52 class gdb_value:
53     __getitem__(x):
54         called by "gdb_value[x]" to access field/index x
55     deref():        dereference gdb_value (if it is a pointer)
56     addrof():       address of gdb_value (if it is an l-value)
57     ... and other methods to query properties of gdb_value ...

```

Fig. 1. EXPOSITOR's Python-based scripting API. The `get_X` and `__len__` methods of `execution` and `trace` are eager, and the remaining methods of those classes return lazy values. Lazy values include `trace`, `snapshot`, and `gdb_value` objects whose contents are computed only on demand and cached.

allow EXPOSITOR to compute individual list elements on-demand in any order, minimizing the number of calls to the time-travel debugger. Furthermore, they shield programmers from the low-level details of controlling the program execution and handling callbacks.

## 2.1 API Overview

Fig. 1 lists the key classes and methods of EXPOSITOR's scripting interface, which is provided as a library inside UndoDB/GDB's Python environment.

2.1.0.1 The `execution` class and the `the_execution` object: The entire execution of the program being debugged is represented by the `execution` class, of which there is a singleton instance named `the_execution`. This class provides several methods for querying the execution. The `get_at(t)`<sup>2</sup> method returns a snapshot object representing the program state at time `t` in the execution (we will describe snapshots in more detail later). Several methods create immutable, sparse projections of the execution, or traces, consisting of program state snapshots at points of interest: the `breakpoints(fn)` and `syscalls(fn)` methods return traces of snapshots at functions and system calls named `fn`, respectively; the `watchpoints(x, rw)` method returns a trace of snapshot when the memory location `x` is read or written; and the `all_calls` and `all_returns` methods return traces of snapshots at all function entries and exits, respectively.

For debugging interactive programs, the `execution` class provides two useful methods: `cont` resumes the execution of the program from when it was last stopped (e.g., immediately after EXPOSITOR is started, or when the program is interrupted by pressing `^C`), and `get_time` gets the latest time of the execution. If a program requires user input to trigger a bug, we often find it helpful to first interrupt the program and call `get_time` to get a reference time, before resuming the execution using `cont` and providing the input trigger.

2.1.0.2 The `trace` class and the `item` class: As mentioned above, the `trace` class represents sparse projections of the execution at points of interest. These traces contain snapshots or other values, indexed by the relevant time in the execution. Initially, traces are created using the execution methods described above, and traces may be further derived from other traces.

The first five trace methods query items in traces. The `tr.__len__()` method is called by the Python built-in function `len(tr)`, and returns the total number of items in the `tr`. The `tr.__iter__()` method is called by Python's `for x in tr` loop, and returns a sequential Python iterator over all items in `tr`. The `get_at(t)` method returns the item at time `t` in the trace, or `None` if there is no item at that time. Since traces are often very sparse, it can be difficult to find items using `get_at`, so the `trace` class also provides two methods, `get_before(t)` and `get_after(t)`, that return the first item found before or after time `t`, respectively, or

2. We use the convention of naming time variables as `t`, and trace variables as `tr`.

`None` if no item can be found. The `get_at`, `get_before`, and `get_after` methods return values that are wrapped in the `item` class, which associates values with a particular point in the execution.

The remaining methods create new traces from existing traces. The `tr.filter(p)` method creates a new trace consisting only of items from `tr` that match predicate `p`. The `tr.map(f)` method creates a new trace of items computed by calling function `f` on each item from `tr`, and is useful for extracting particular values of interest from snapshots. The `tr.slice(t0, t1)` method creates a new trace that includes only items from `tr` between times `t0` and `t1`.

The trace class also provides several more complex methods to derive new traces. Three methods create new traces by merging traces. First, `tr0.merge(f, tr1)` creates a new trace containing the items from both `tr0` and `tr1`, calling function `f` to combine any items from `tr0` and `tr1` that occur at the same time (Fig. 2a). `None` can be passed for `f` if `tr0` and `tr1` contain items that can never coincide, e.g., if `tr0` contains calls to `foo` and `tr1` contains calls to `bar`, since `f` will never be called in this case. Next, `tr0.trailing_merge(f, tr1)` creates a new trace by calling `f` to merge each item from `tr0` with the immediately preceding item from `tr1`, or `None` if there is no preceding item (Fig. 2b). Lastly, `rev_trailing_merge` is similar to `trailing_merge` except that it merges with future items rather than past items (Fig. 2c).

The remaining four methods create new traces by computing over prefixes or suffixes of an input trace. The `scan` method performs a fold- or reduce-like operation for every prefix of an input trace (Fig. 2d). It is called as `tr.scan(f, acc)`, where `f` is a binary function that takes an accumulator and an item as arguments, and `acc` is the initial accumulator. It returns a new trace containing the same number of items at the same times as in the input trace `tr`, where the  $n$ th output item  $out_n$  is recursively computed as:

$$out_n = \begin{cases} in_n \textcircled{f} out_{n-1} & \text{if } n > 0 \\ in_n \textcircled{f} acc & \text{if } n = 0 \end{cases}$$

where `f` is written infix as  $\textcircled{f}$ . The `rev_scan` method is similar, but deriving a trace based on future items rather than past items (Fig. 2e). `rev_scan` computes the output item  $out_n$  as follows:

$$out_n = \begin{cases} in_n \textcircled{f} out_{n+1} & \text{if } 0 \leq n < length - 1 \\ in_n \textcircled{f} acc & \text{if } n = length - 1 \end{cases}$$

Lastly, `tscan` and `rev_tscan` are variants of `scan` and `rev_scan`, respectively, that take an associative binary function but no accumulator, and can sometimes be more efficient. These two methods are described in Sec. 3.2.

### 2.1.0.3 The `snapshot` class and the `gdb_value` class:

The `snapshot` class represents a program state at a particular point in time and provides methods for accessing that state, e.g., `read_var(x)` returns the value of a variable named `x` in the current stack frame, `read_retaddrs` returns

the list of return addresses on the stack, `backtrace` prints the stack backtrace, and so on.

The `gdb_value` class represents values in the program being debugged at a particular point in time, and provides methods for querying those values. For example, `v.__getitem__(x)` is called by the Python indexing operator `v[x]` to access `struct` fields or array elements, `deref` dereferences pointer values, `addrOf` returns the address of an l-value, and so forth. These `gdb_value` objects are automatically coerced to the appropriate Python types when they are compared against built-in Python values such as ints, for example; it is also sometimes useful to manually coerce `gdb_value` objects to specific Python types, e.g., to treat a pointer value as a Python int.

Both the `snapshot` and `gdb_value` classes are thin wrappers around GDB's Python API and UndoDB. When a method of a `snapshot` or `gdb_value` object is called, it first directs UndoDB to jump to the point in time in the program execution that is associated with the object. Then, it calls the corresponding GDB API, wrapping the return value in `snapshot` and `gdb_value` as necessary. In general, the semantics of `snapshot` and `gdb_value` follows GDB, e.g., the EXPOSITOR's notion of stack frames is based on GDB's notion of stack frames. We also provide several methods, such as `read_retaddrs`, that return the result of several GDB API calls in a more convenient form. Additionally, all of GDB's Python API is available and may be used from within EXPOSITOR.

Given a debugging hypothesis, we use the EXPOSITOR interface to apply the following recipe. First, we call methods on the `_execution` to derive one or more traces that contain events relevant to the hypothesis; such events could be function calls, breakpoints, system calls, etc. Next, we combine these traces as appropriate, applying trace methods such as `filter`, `map`, `merge`, and `scan` to derive traces of properties predicted by our hypothesis. Finally, we query the traces using methods such as `get_before` and `get_after` to find evidence of properties that would confirm or refute our hypothesis.

## 2.2 Warm-up Example: Examining `foo` Calls in EXPOSITOR

To begin with a simple example, let us consider the task of counting the number of calls to a function `foo`, to test a hypothesis that an algorithm is running for an incorrect number of iterations, for example. Counting `foo` calls takes just two lines of code in EXPOSITOR. We first use the `breakpoints` method of the `_execution` to create the `foo` trace:

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

This gives us a trace containing all calls to `foo`. We can then count the calls to `foo` using the Python `len` function, and print it:

```
60 print len(foo)
```

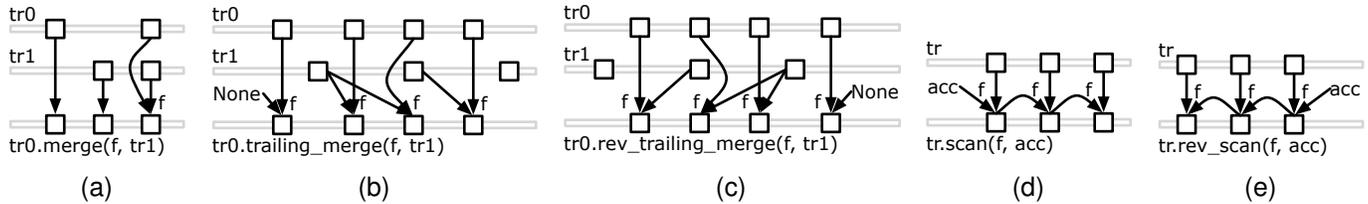


Fig. 2. Illustration of complex trace operations.

Later, we may want to count only calls to `foo(x)` where `x == 0`, perhaps because we suspect that only these calls are buggy. We can achieve this using the `filter` method on the `foo` trace created above:

```
61 foo_0 = foo.filter(lambda snap: snap.read_var("x") == 0)
```

Here, we call `filter` with a predicate function that takes a snapshot object (at calls to `foo`), reads the variable named `x`, and returns whether `x == 0`. The resulting trace contains only calls to `foo(0)`, which we assign to `foo_0`. We can then count `foo_0` as before:

```
62 print len(foo_0)
```

After more investigation, we may decide to examine calls to both `foo(0)` and `foo(1)`, e.g., to understand the interaction between them. We can create a new trace of `foo(1)` calls, and merge it with `foo(0)`:

```
63 foo_1 = foo.filter(lambda snap: snap.read_var("x") == 1)
64 foo_01 = foo_0.merge(None, foo_1)
```

We define `foo_1` just like `foo_0` but with a different predicate. Then, we use the `merge` method to merge `foo_0` and `foo_1` into a single trace `foo_01` containing calls to both `foo(0)` and `foo(1)`, passing `None` for the merging function as `foo(0)` and `foo(1)` can never coincide. Note that we are able to reuse the `foo` and `foo_0` traces in a straightforward manner; under the hood, EXPOSITOR will also have cached the computation of `foo` and `foo_0` from the earlier and reuse them here.

Finally, we may want to take a closer look at the very first call to either `foo(0)` or `foo(1)`, which we can do using the `get_after` method:

```
65 first_foo_01 = foo_01.get_after(0)
```

We call `get_after` on `foo_01` to find the first item after time 0, i.e., the beginning of the execution, that contains the snapshot of a `foo(0)` or `foo(1)` call.

In this example, observe how we began with a simple debugging task that counts all calls to `foo` in a trace to answer our initial hypothesis, then gradually create more traces or combined existing ones and queried them as our hypothesis evolves. EXPOSITOR is particularly suited for such incremental, interactive style of debugging.

### 2.2.1 Comparison to GDB's Python API

In contrast to EXPOSITOR, it takes 16 lines of code to count `foo` calls using GDB's standard Python API, as shown below:

```
66 count = 0; more = True
67 foo = gdb.Breakpoint("foo")
68 def stop_handler(evt):
69     if isinstance(evt, gdb.BreakpointEvent) \
70         and foo in evt.breakpoints:
71         global count; count += 1
72 def exit_handler(evt):
73     global more; more = False
74 gdb.events.stop.connect(stop_handler)
75 gdb.events.exited.connect(exit_handler)
76 gdb.execute("start")
77 while more:
78     gdb.execute("continue")
79 gdb.events.exited.disconnect(exit_handler)
80 gdb.events.stop.disconnect(stop_handler)
81 foo.delete()
```

On line 66, we first initialize two variables, `count` and `more`, that will be used to track of the number of calls to `foo` and to track if the execution has ended respectively. Then, on line 67, we create a breakpoint at the call to `foo`. Next, we create a callback function named `stop_handler` on lines 68–71 to handle breakpoint events. In this function, we first check on lines 69–70 to see if the breakpoint triggered is the one that we have set, and if so, we increment `count` on line 71. We also create a callback function named `exit_handler` on lines 72–73 to handle stop events which are fired when the program execution ends. This function simply resets the `more` flag when called.

After that, we register `stop_handler` and `exit_handler` with GDB on lines 74–75, and start the program execution on line 76. GDB will run the program until it hits a breakpoint or the end of the execution is reached, calling `stop_handler` in the former case, or `exit_handler` in the latter case. Then, we enter a loop on lines 77–78 that causes GDB to continue running the program until `more` is `False`, i.e., the program has exited. Once that happens, we deregister the event handlers from GDB and delete the breakpoint on lines 79–81, cleaning up after ourselves to ensure that the callbacks and breakpoint will not be unintentionally triggered by other scripts.

It also takes more work to refine this GDB script to answer other questions about `foo`, compared to EXPOSITOR traces. For example, to count calls to `foo(0)`, we would have to modify the GDB script to add the `x == 0` predicate and rerun it, instead of simply calling `filter` on the `foo` trace. As another example, if we were given two different scripts, one that counts `foo(0)` and another that counts `foo(1)`, it would be difficult to combine those scripts as they each contain their own driver loops and shared

variables; it would be easier to just modify one of those script than to attempt to reuse both. In contrast, it took us one line to use the merge method to combine the `foo_0` and `foo_1` traces. Finally, note that EXPOSITOR caches and reuses trace computation automatically, whereas we would need some foresight to add caching to the GDB script in a way that can be reused by other scripts.

### 2.3 Example: Reverse Engineering a Stack-Smashing Attack

We now illustrate the use of EXPOSITOR with a more sophisticated example: reverse engineering a stack-smashing attack, in which malware overflows a stack buffer in the target program to overwrite a return address on the stack, thereby gaining control of the program counter [7].

We develop a reusable script that can detect when the stack has been smashed in any program, which will help pinpoint the attack vector. Our script maintains a *shadow stack* of return addresses and uses it to check that only the top of the stack is modified between function calls or returns; any violation of this property indicates the stack has been smashed.

We begin by using the `all_calls` and `all_returns` methods on `the_execution` to create traces of just the snapshots at function calls and returns, respectively:

```
82 calls = the_execution.all_calls()
83 rets = the_execution.all_returns()
```

Next, we use `merge` to combine these into a single trace, passing `None` for the merging function as function calls and returns can never coincide. We will use this new trace to compare consecutive calls or returns:

```
84 calls_rets = calls.merge(None, rets)
```

Now, we map over `call_returns` to apply the `read_retaddr` method, returning the list of return addresses on the call stack. This creates a trace of shadow stacks at every call and return:

```
85 shadow_stacks = calls_rets.map(
86     lambda s: map(int, s.read_retaddr()))
```

We also use `map` to coerce the return addresses to Python ints.

Then we need to check that, between function calls and returns, the actual call stack matches the shadow stack except for the topmost frame (one return address may be added or removed). We use the following function:

```
87 def find_corrupted(ss, opt_shadow):
88     if opt_shadow.force() is not None:
89         for x, y in zip(ss.read_retaddr(), opt_shadow.force()):
90             if int(x) != y:
91                 return x # l-value of return address on stack
92     return None
```

Here, `find_corrupted` takes as arguments a snapshot `ss` and its immediately preceding shadow stack `opt_shadow`; the `opt_` prefix indicates that there may not be a prior

shadow stack (if `ss` is at the first function call), and we need to call the `force` method on `opt_shadow` to retrieve its value (we will explain the significance of this in Sec. 3). If there is a prior shadow stack, we compare every return address in `ss` against the shadow stack and return the first location that differs, or `None` if there are no corrupted addresses. (The `zip` function creates a list of pairs of the respective elements of the two input lists, up to the length of the shorter list.)

Finally, we generate a trace of corrupted memory locations using the `trailing_merge` method, calling `find_corrupted` to merge each function call and return from `call_rets` with the immediately preceding shadow stack in `shadow_stacks`. We filter `None` out of the result:

```
93 corrupted_addrs = calls_rets \
94     .trailing_merge(find_corrupted, shadow_stacks) \
95     .filter(lambda x: x is not None)
```

The resulting trace contains exactly the locations of corrupted return addresses at the point they are first evident in the trace.

### 2.4 Mini Case Study: Running EXPOSITOR on tinyhttpd

We used the script just developed on a version of `tinyhttpd` [8] that we had previously modified to include a buffer overflow bug. We created this version of `tinyhttpd` as an exercise for a security class in which students develop exploits of the vulnerability.

As malware, we deployed an exploit that uses a return-to-libc attack [9] against `tinyhttpd`. The attack causes `tinyhttpd` to print “Now I pwn your computer” to the terminal and then resume normal operation. Finding buffer overflows using standard techniques can be challenging, since there can be a delay from the exploit overflowing the buffer to the payload taking effect, during which the exploited call stack may be erased by normal program execution. The payload may also erase evidence of itself from the stack before producing a symptom.

To use EXPOSITOR, we call the `expositor` launcher with `tinyhttpd` as its argument, which will start a GDB session with EXPOSITOR’s library loaded, and then enter the Python interactive prompt from GDB:<sup>3</sup>

```
96 % expositor tinyhttpd
97 (expositor) python-interactive
```

Then, we start running `tinyhttpd`:

```
98 >>> the_execution.cont() # start running
99 httpd running on port 47055
```

When `tinyhttpd` launches, it prints out the port number on which it accepts client connections. On a different terminal, we run the exploit with this port number:

```
100 % ./exploit.py 47055
101 Trying port 47055
102 pwning...
```

3. GDB contains an existing python command that is not interactive; `python-interactive` is a new command that we have submitted to GDB, and is available as of GDB 7.6.

At this point, tinyhttpd prints the exploit message, so we interrupt the debugger and use EXPOSITOR to find the stack corruption, starting from the time when we interrupted it:

```

103 Now I pwn your computer
104 ^C
105 Program received signal SIGINT, Interrupt
106 >>> corrupted_addrs = stack_corruption()
107           # function containing Sec. 2.3 code
108 >>> time = the_execution.get_time()
109 >>> last_corrupt = corrupted_addrs.get_before(time)

```

Items in a trace are indexed by time, so the `get_before` method call above tells EXPOSITOR to start computing `corrupted_addrs` from the interrupted time backward and find the first function call or return when the stack corruption is detected. We can print the results:

```

110 >>> print time
111 56686.8
112 >>> print last_corrupt
113 Item(56449.2, address)

```

This shows that the interrupt occurred at time 56686.8, and the corrupted stack was first detected at a function call or return at time 56449.2. We can then find and print the snapshot that corrupted the return address with:

```

114 >>> bad_writes = the_execution \
115           .watchpoints(last_corrupt.value, rw=WRITE)
116 >>> last_bad_write = bad_writes.get_before(last_corrupt.time)
117 >>> print last_bad_write
118 Item(56436.0, snapshot)

```

We find that the first write that corrupted the return address occurred at time 56436.0. We can then inspect the snapshot via `last_bad_write.value`. In this case, the backtrace of the very first snapshot identifies the exact line of code in tinyhttpd that causes the stack corruption—a socket rcv with an out-of-bounds pointer. Notice that to find the bug, EXPOSITOR only inspected from time 56686.8 to time 56436.0. Moreover, had `last_corrupt` not explained the bug, we would then call `corrupted_addrs.get_before(last_corrupt.time)` to find the prior corruption event, inspecting only as much of the execution as needed to track down the bug.

This mini case study also demonstrates that, for some debugging tasks, it can be much faster to search backward in time. It takes only 1 second for `corrupted_addrs.get_before(time)` to return; whereas if we had instead searched forward from the beginning (e.g., simulating a debugger without time-travel):

```

119 first_corrupted = corrupted_addrs.get_after(0)

```

it takes 4 seconds for the answer to be computed. Using EXPOSITOR, users can write scripts that search forward or backward in time, as optimal for the task.

### 3 LAZY TRACES IN EXPOSITOR

As just discussed, EXPOSITOR allows users to treat traces as if they were lists of snapshots. However, for many applications it would be impractical to eagerly record

and analyze full program snapshots at every program point. Instead, EXPOSITOR uses the underlying time-travel debugger, UndoDB, to construct snapshots on demand and to discard them when they are no longer used (since it is expensive to keep too many snapshots in memory at once). Thus the major challenge is to minimize the demand for snapshots, which EXPOSITOR accomplishes by constructing and manipulating traces *lazily*.

More precisely, all of the trace generators and combinators, including `execution.all_calls`, `trace.map`, `trace.merge`, etc., return immediately without invoking UndoDB. It is only when final values are demanded, with `execution.get_at`, `trace.get_at`, `trace.get_after`, or `trace.get_before`, that EXPOSITOR queries the actual program execution, and it does so only as much as is needed to acquire the result. For example, the construction of `corrupted_addrs` in Sec. 2.4, line 106 induces *no* time travel on the underlying program—it is not until the call to `corrupted_addrs.get_before(time)` in Sec. 2.4, line 109 that EXPOSITOR uses the debugger to acquire the final result.

To achieve this design, EXPOSITOR uses a lazy, interval-tree-like data structure to implement traces. More precisely, a trace is a binary tree whose nodes are annotated with the (closed) lower-bound and (open) upper-bound of the time intervals they span, and leaf nodes either contain a value or are empty. The initial tree for a trace contains no elements (only its definition), and EXPOSITOR materializes tree nodes as needed.

As a concrete example, the following trace constructs the tree shown on the right, with a single lazy root node spanning the interval  $[0, \infty)$ , which we draw as a dotted box and arrow.

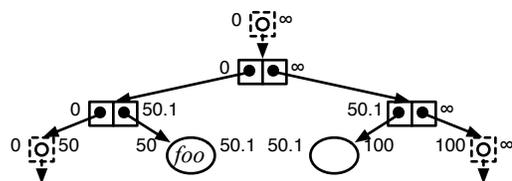
```

120 foo = the_execution.breakpoints("foo")

```



Now suppose we call `foo.get_before(100)`. EXPOSITOR sees that the query is looking for the last call to `foo` before time 100, so it will ask UndoDB to jump to time 100 and then run backward until hitting such a call. Let us suppose the call is at time 50, and the next instruction after that call is at time 50.1. Then EXPOSITOR will expand the root node shown above to the following tree:



Here the trace has been subdivided into four intervals: The intervals  $[0, 50)$  and  $[100, \infty)$  are lazy nodes with no further information, as EXPOSITOR did not look at those portions of the execution. The interval  $[50, 50.1)$  contains the discovered call, and the interval  $[50.1, 100)$  is fully resolved and contains no calls to `foo`. Notice that if we ask the same query again, EXPOSITOR can traverse the

interval tree above to respond without needing to query UndoDB.

Likewise, calling `get_at(t)` or `get_after(t)` either returns immediately (if the result has already been computed) or causes UndoDB to jump to time `t` (and, for `get_after(t)`, to then execute forward). These methods may return `None`, e.g., if a call to `foo` did not occur before/after/at time `t`.

As our micro-benchmarks in Sec. 5 will show, if we request about 10–40% of the items in a trace, computing traces lazily takes less time than computing eagerly, depending on the query pattern as well as the kind of computations done. This makes lazy traces ideal for debugging tasks where we expect programmers to begin with some clues about the location of the bug. For example, we start looking for stack corruption from the end of the execution in Sec. 2.4, line 109, because stack corruptions typically occur near the end of the execution.

### 3.1 Lazy Trace Operations

We implement `filter` and `map` lazily on top of the interval tree data structure. For a call `tr1 = tr0.map(f)`, we initially construct an empty interval tree, and when values are demanded in `tr1` (by `get_X` calls), EXPOSITOR conceptually calls `tr0.get_X`, applies `f` to the result, and caches the result for future use. Calls to `tr0.filter(p)` are handled similarly, constructing a lazy tree that, when demanded, repeatedly gets values from `tr0` until `p` is satisfied. Note that for efficiency, EXPOSITOR’s does not actually call `get_X` on the root node of `tr0`; instead, it directly traverses the subtree of `tr0` corresponding to the uninitialized subtree of the derived trace.

The implementation of `tr0.merge(f, tr1)` also calls `get_X` on `tr1` as required. For a call `tr.slice(t0, t1)` EXPOSITOR creates an interval tree that delegates `get_X` calls to `tr`, asking for items from time `t0` to time `t1`, and returns `None` for items that fall outside that interval.

For the last four operations, `[rev_]trailing_merge` and `[rev_]scan`, EXPOSITOR employs additional laziness in the helper function argument `f`. To illustrate, consider a call to `tr.scan(f, acc)`. Here, EXPOSITOR passes the accumulator to `f` wrapped in an instance of class `lazy`, defined as follows:

```

121 class lazy:
122     force():      return the actual value
123     is_forced():  return whether force has been called

```

The `force` method, when first called, will compute the actual value and cache it; the cached value is returned in subsequent calls. Thus, `f` can force the accumulator as needed, and if it is not forced, it will not be computed.

To see the benefit, consider the following example, which uses `scan` to derive a new trace in which each item is a count of the number of consecutive calls to `foo` with nonzero arguments, resetting the count when `foo` is called with zero:

```

124 foo = execution.breakpoints("foo") # void foo(int x)
125 def count_nonzero_foo(lazy_acc, snapshot):
126     if snapshot.read_var("x") != 0:

```

```

127         return lazy_acc.force() + 1
128     else:
129         return 0
130 nonzero_foo = foo.scan(count_nonzero_foo, 0)

```

Notice that if `lazy_acc` were not lazy, EXPOSITOR would have to compute its value before calling `count_nonzero_foo`. By the definition of `scan` (Fig. 2d), this means that it must recursively call `count_nonzero_foo` to compute all prior output items before computing the current item, even if it is unnecessary to do so, e.g., if we had called `nonzero_foo.get_before(t)`, and the call to `foo` just before time `t` had argument `x=0`. Thus, a lazy accumulator avoids this unnecessary work. EXPOSITOR uses a lazy accumulator in `rev_scan` for the same reason.

Likewise, observe that in `tr0.trailing_merge(f, tr1)`, for a particular item in `tr0` the function `f` may not need to look in `tr1` to determine its result; thus, EXPOSITOR wraps the `tr1` argument to `f` in an instance of class `lazy`. The implementation of `rev_trailing_merge` similarly passes lazy items from `tr1` to `f`. Note that there is no such laziness in the regular merge operation. The reason is that in `tr0.merge(f, tr1)`, the items from `tr0` and `tr1` that are combined with `f` occur at the same time. Thus, making `f`’s arguments lazy would not reduce demands on the underlying time-travel debugger.

### 3.2 Tree Scan

Finally, EXPOSITOR provides another list combinator, *tree-scan*, which is a lazier variant of `scan` that is sometimes more efficient. The `tscan` method computes an output for every prefix of an input trace by applying an associative binary function in a tree-like fashion (Fig. 3a). It is invoked with `tr.tscan(f)`, where `f` must be an associative function that is lazy and optional in its left argument and lazy in its right argument. The `tscan` method generates an output trace of the same length as the input trace, where the  $n$ th output  $out_n$  is defined as:

$$out_n = in_0 \circledast in_1 \circledast \dots \circledast in_n$$

where  $\circledast$  is written infix as  $\circledast$ . Notice that there is no accumulator, and EXPOSITOR can apply  $f$  in any order, since it is associative. When a value at time  $t$  is demanded from the output trace, EXPOSITOR first demands the item  $in_n$  at that time in the input trace (if no such item exists, then there is no item at that time in the output trace). Then EXPOSITOR walks down the interval tree structure of the input trace, calling  $f$  (only if demanded) on each internal tree node’s children to compute  $out_n$ . Since the interval tree for the input trace is computed lazily,  $f$  may sometimes be called with `None` as a left argument, for the case when  $f$  forces an interval that turns out to contain no values; thus for correctness, we also require that  $f$  treats `None` as a left identity. (The right argument corresponds to  $in_n$  and so will never be `None`.)

Because both arguments of  $f$  are lazy, EXPOSITOR avoids computing either argument unnecessarily. The

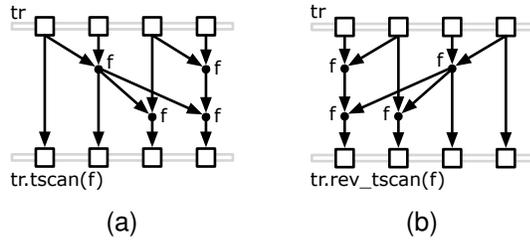


Fig. 3. Illustration of tree-scan operations.

is\_forced method of the lazy class is particularly useful for tscan, as it allows us to determine if either argument has been forced, and if so, evaluate the forced argument first. For example, we can check if a trace contains a true value as follows:

```

131 def has_true(lazyleft, lazyright):
132     return lazyleft.is_forced() and lazyleft.force() \
133         or lazyright.is_forced() and lazyright.force() \
134         or lazyleft.force() or lazyright.force()
135 has_true_trace = some_trace.tscan(has_true)
136 last_has_true = has_true_trace.get_before("inf")

```

The best case for this example occurs if either lazyleft or lazyright have been forced by a prior query, in which case either the first clause (line 132) or second clause (line 133) will be true and the unforced argument need not be computed due to short-circuiting.

EXPOSITOR's rev\_tscan derives a new trace based on future items instead of past items (Fig. 3b), computing output item  $out_n$  as:

$$out_n = in_n \textcircled{f} in_{n+1} \textcircled{f} \cdots \textcircled{f} in_{length-1}$$

Here, the right argument to  $f$  is optional, rather than the left.

## 4 THE EDIT HASH ARRAY MAPPED TRIE

Many of the EXPOSITOR scripts we have written use sets or maps to record information about the program execution. For example, in Sec. 1, we suggested the use of a shadow set to debug the implementation of a custom set data structure. Unfortunately, a typical eager implementation of sets or maps could demand all items in the traces, defeating the intention of EXPOSITOR's lazy trace data structure. To demonstrate this issue, consider the following code, which uses Python's standard (non-lazy) set class to collect all arguments in calls to a function foo:

```

137 foos = the_execution.breakpoints("foo") # void foo(int arg)
138 def collect_foo_args(lazy_acc, snap):
139     return lazy_acc.force().union( \
140         set([ int(snap.read_var("arg")) ]))
141 foo_args = foos.scan(collect_foo_args, set())

```

Notice that we must force lazy\_acc to call the union method which will create a deep copy of the updated set (lines 139–140). Unfortunately, forcing lazy\_acc causes

```

142 class edithamt:
143     # lookup methods
144     contains(k):
145                                     Return if key k exists
146     find(k):
147                                     Return the latest value for k or None if not found
148     find_multi(k):
149                                     Return an iterator of all values bound to k
150
151     # static factory methods to create new EditHAMTs
152     empty():
153                                     Create an empty EditHAMT
154     add(lazy_ah, k):
155                                     Add binding of k to itself to lazy_ah
156     addkeyvalue(lazy_ah, k, v):
157                                     Add binding of k to v to lazy_ah
158     remove(lazy_ah, k):
159                                     Remove all bindings of k from lazy_ah
160     removeone(lazy_ah, k):
161                                     Remove the latest binding of k to any value from lazy_ah
162     removekeyvalue(lazy_ah, k, v):
163                                     Remove the latest binding of k to v from lazy_ah
164     concat(lazy_ah1, lazy_ah2):
165                                     Concatenate lazy_ah2 edit history to lazy_ah1

```

Fig. 4. The EditHAMT API.

the immediately preceding set to be computed by recursively calling collect\_foo\_args. As a result, we must compute all preceding sets in the trace even if a particular query could be answered without doing so.

To address these problems, we developed the *edit hash array mapped trie* (EditHAMT), a new set, map, multiset, and multimap data structure that supports lazy construction and queries. The EditHAMT complements the trace data structure; as we will explain, and our micro-benchmark in Sec. 5.4 will show, the EditHAMT can be used in traces without compromising trace laziness, unlike eager sets or maps.

### 4.1 EditHAMT API

From the user's perspective, the EditHAMT is an immutable data structure that maintains the entire history of edit operations for each EditHAMT. Fig. 4 shows the EditHAMT API. The edithamt class includes contains(k) to determine if key k exists, and find(k) to look up the latest value mapped to key k. It also includes the find\_multi(k) method to look up all values mapped to key k, returned as a Python iterator that incrementally looks up each mapped value. EditHAMT operations are implemented as static factory methods that create new EditHAMTs. Calling edithamt.empty() creates a new, empty EditHAMT. Calling edithamt.add(lazy\_ah, k) creates a new EditHAMT by adding to lazy\_ah, the prior EditHAMT, a binding from key k to itself (treating the EditHAMT as a set or multiset). Similarly, edithamt.addkeyvalue(lazy\_ah, k, v) creates a new EditHAMT by adding to lazy\_ah a binding from key k value v (treating the EditHAMT as a map or multimap). Conversely, calling edithamt.remove(lazy\_ah, k) creates a new EditHAMT by removing

all bindings of key  $k$  from `lazy_eh`. Lastly, calling `edithamt.removeone(lazy_eh, k)` or `edithamt.removekeyvalue(lazy_eh, k, v)` creates new EditHAMTs by removing from `lazy_eh` the most recent binding of key  $k$  to any value or to a specific value  $v$ . The `lazy_eh` argument to these static factory methods is `lazy` so that we need not force it until a call to `contains`, `find` or `find_multi` demands a result. For convenience, the `lazy_eh` argument can also be `None`, which is treated as an empty EditHAMT.

The last static factory method, `edithamt.concat(lazy_eh1, lazy_eh2)`, concatenates the edit histories of its arguments. For example:

```
166 eh_rem = edithamt.remove(None, "x")
167 eh_add = edithamt.addkeyvalue(None, "x", 42)
168 eh = edithamt.concat(eh_add, eh_rem)
```

Here `eh` is the empty EditHAMT, since it contains the additions in `eh_add` followed by the removals in `eh_rem`. A common EXPOSITOR script pattern is to map a trace to a sequence of EditHAMT additions and removals, and then use `edithamt.concat` with `scan` or `tscan` to concatenate those edits.

## 4.2 Example: EditHAMT to Track Reads and Writes to a Variable

As an example of using the EditHAMT, we present one piece of the race detector used in our Firefox case study (Sec. 6). The detector compares each memory access against prior accesses to the same location from any thread. Since UndoDB serializes thread schedules, each read need only be compared against the immediately preceding write, and each write against the immediately preceding write as well as reads between the two writes.

We use the EditHAMT as a multimap in the following function to track the access history of a given variable  $v$ :

```
169 def access_events(v):
170     reads = the_execution.watchpoints(v, rw=READ) \
171         .map(lambda s: edithamt.addkeyvalue( \
172             None, v, ("read", s.get_thread_id())))
173     writes = the_execution.watchpoints(v, rw=WRITE) \
174         .map(lambda s: edithamt.addkeyvalue( \
175             edithamt.remove(None, v), \
176             v, ("write", s.get_thread_id())))
177     return reads.merge(None, writes)
```

In `access_events`, we create the trace reads by finding all reads to  $v$  using the `watchpoints` method (line 170), and then mapping each snapshot to a singleton EditHAMT that binds  $v$  to a tuple of "read" and the running thread ID (lines 171–172). Similarly, we create the trace writes for writes to  $v$  (line 173), but instead map each write snapshot to an EditHAMT that first removes all prior bindings for  $v$  (line 175), then binds  $v$  to a tuple of "write" and the thread ID (lines 174–176). Finally, we merge reads and writes, and return the result (line 177).

We are not done yet, since the EditHAMTs in the trace returned by `access_events` contain only edit operations corresponding to individual accesses to  $v$ . We can get an EditHAMT trace that records all accesses to  $v$  from the

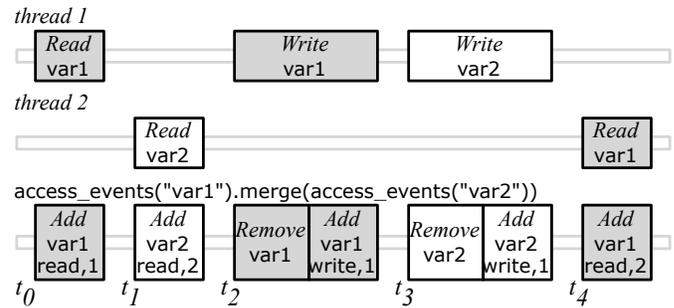


Fig. 5. Example execution with two threads accessing `var1` (gray) and `var2`, and the corresponding EditHAMT operations returned by `access_events`.

beginning of the execution by using `scan` with `edithamt.concat` to concatenate the individual EditHAMTs. For example, we can record the access history of `var1` as follows:

```
178 var1_history = access_events("var1").scan(edithamt.concat)
```

We can also track multiple variables by calling `access_events` on each variable, merging the traces, then concatenating the merged trace, e.g., to track `var1` and `var2`:

```
179 access_history = \
180     access_events("var1").merge(access_events("var2")) \
181     .scan(edithamt.concat)
```

Since trace methods are lazy, this code completes immediately; the EditHAMT operations will only be applied, and the underlying traces forced, when we request a particular access, e.g., at the end of the execution (time "inf"):

```
182 last = access_history.get_before("inf")
```

To see laziness in action, consider applying the above analysis to an execution depicted in Fig. 5, which shows two threads at the top and the corresponding EditHAMT operations at the bottom. Suppose we print the latest access to `var1` at time  $t_4$  using the `find` method:

```
183 >>> print last.find("var1")
184 ("read", 2)
```

Because "var1" was just added at time  $t_4$ , answering this query will only force the EditHAMT and query the time-travel debugger at time  $t_4$ , and not before.

As another example, suppose we want to find all accesses to `var1` from the last access backward using `find_multi`:

```
185 >>> for mem_access in last.find_multi("var1"):
186     print mem_access
187 ("read", 2)
188 ("write", 1)
```

Here since all "var1" bindings added prior to time  $t_2$  were removed at time  $t_2$ , the results are computed without forcing any EditHAMTs or querying the debugger before time  $t_2$ .

### 4.3 Implementation

The EditHAMT is inspired by the *hash array mapped trie* (HAMT) [10]. Like the HAMT, the EditHAMT is a hybrid data structure combining the fast lookup of a hash table and the memory efficiency of a trie. The HAMT is a hash-based data structure built in a manner analogous to a hash table. Whereas a hash table uses a bucket array to map keys to values, the HAMT uses an *array mapped trie* (AMT)—a trie that maps fixed-width integer keys to values—for the same purpose. When a hash collision occurs, the HAMT resolves the collision by replacing the colliding entry with a nested HAMT, rehashing the colliding keys, and inserting those keys in the nested HAMT using the new hash values.

We developed the EditHAMT by making two changes to the traditional HAMT. First, we replaced the AMT with the *LazyAMT*, which supports lazy, rather than eager, updates. Second, we resolve hash collisions, as well as support remove and multiset/multimap operations, using *EditLists*, which are lazy linked-lists of nodes tallying edit operations on the EditHAMT; the tails are lazily retrieved from the prior EditHAMT.

#### 4.3.1 LazyAMT: Lazy Array Mapped Tries

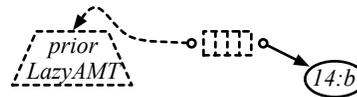
The first piece of the EditHAMT is the LazyAMT, which is a lazy, immutable variant of the AMT that maps fixed-width integer keys of size  $k$  bits to values. We implement the LazyAMT using *lazy sparse arrays* of size  $2^w$  as internal nodes, where  $w$  is the bit-width of the array index such that  $w < k$ , and store key-value bindings as leaf nodes. We will divide the key into  $w$ -bit words, where each  $w$ -bit word is used to index an internal node during a lookup; the key can be padded as necessary if  $k$  is not a multiple of  $w$ .

Lazy sparse arrays combine the properties of lazy values and sparse arrays: each element of a lazy sparse array is computed and cached when first indexed, akin to forcing a lazy value, and null elements are stored compactly, like sparse arrays. We implement lazy sparse arrays using two bitmaps to track which elements are initialized and non-null,<sup>4</sup> respectively, and an array to store initialized, non-null elements.

To build the EditHAMT, we need to support two operations on the LazyAMT: adding a key-value binding to a LazyAMT, and merging two LazyAMTs into a single LazyAMT. We will explain how the LazyAMT works by example, using an internal node index bit-width of  $w = 2$  bits and a key size of  $k = 6$  bits.

4.3.1.1 Adding a key-value binding: When we add a binding such as  $14 : b$  to a LazyAMT, we first create a new lazy sparse array representing the root node:

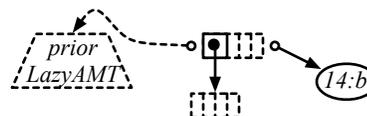
4. It is more efficient to track non-null elements as many modern processors provide a POPCNT instruction, which counts the number of 1 bits in a word, that can be used to compute the index of a non-null element in the storage array.



Initially, all elements of the root node are uninitialized, which we depict as four narrow dotted boxes; we will use the convention of numbering the boxes from left to right, i.e., in binary, the leftmost box is element 00, and the rightmost box is element 11. In addition, we also maintain a lazy reference to the prior LazyAMT; we do not yet need to know what the prior LazyAMT contains, which we indicate with a dotted arrow. In fact, the lazy reference allows us to further defer the construction of (the root node of) the prior LazyAMT, i.e., the prior LazyAMT may not exist yet when we add the binding  $14 : b$ ; we indicate this with a dotted arrow. For example, in EXPOSITOR, we may query UndoDB to determine what binding should be added only when the lazy reference to the prior LazyAMT is first forced. We also need to store the binding  $14 : b$ , to be added when the LazyAMT is sufficiently initialized.

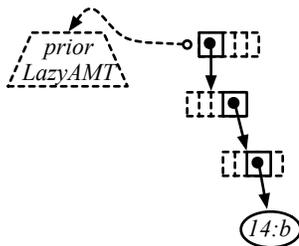
The actual construction of the LazyAMT occurs only when we look up a binding. The lookup is a standard trie lookup, however, since internal nodes are lazy sparse arrays, the elements of those arrays will be initialized as necessary when we access those elements during the lookup. We initialize an element in one of three ways, depending on whether that element is along the lookup path of the binding we previously set aside to be added, and whether we have reached the end of the lookup path. If the element is along the lookup path the binding and we have not reached the end of the lookup path, we create the next internal node and initialize the element to that node. If the element is along the lookup path and we have reached the end of the lookup path, we initialize the element to a leaf node containing that binding. Otherwise, if the element is not along the lookup path of the binding, we initialize it to point to the same subtree as the corresponding element (at the same partial lookup path) in the prior LazyAMT, or to be null if the corresponding element does not exist. Note that in the last case, prior LazyAMTs will be recursively initialized as necessary.

For example, suppose that we look up the binding for key 14. First, we split up the key into  $w$ -bit words, here, 001110 in binary; this is the lookup path for key 14. Then, we use the first word, 00, to index the root node. We need to initialize the element at 00 as this is the first time we accessed it. Since this particular LazyAMT was created by adding  $14 : b$  and the 00 element of the root node is along the lookup path for key 14, we initialize that element to a new uninitialized internal node below the root node:



Here, we depict the initialized element of the root node as a square unbroken box, and a non-lazy reference to the just created internal node as an unbroken arrow.

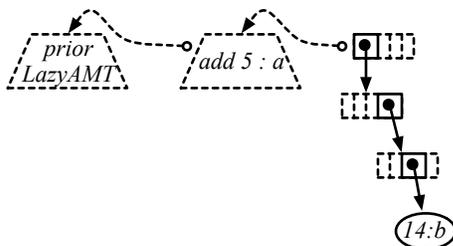
We continue the lookup by using the next word, 11, to index the just created internal node. Since 11 is again along the lookup path, we initialize the corresponding element to another internal node. We repeat the process again with the last word, 10, but now that we have exhausted all bits in the lookup key, we initialize the element for 10 to point to a leaf node containing the binding  $14 : b$  that we previously set aside. This results in a partially initialized LazyAMT:



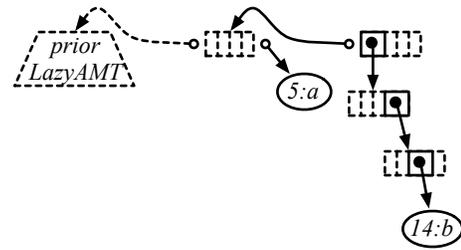
We finish the lookup for key 14 and return  $b$ .

The example so far illustrates how the lookup process drives the initialization process in an interleaved manner. Also, since we are looking up a key that was just inserted into the LazyAMT, we did not need to refer to the prior LazyAMT at all. These properties allow LazyAMTs to be constructed lazily, by initializing only as much as necessary to answer lookup queries.

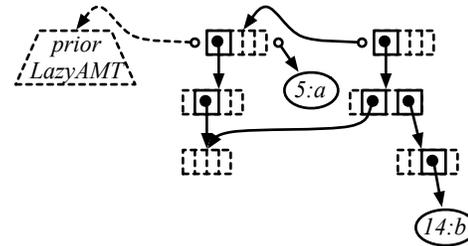
We continue the example by considering the case of looking up a key that was not added by the most recent LazyAMT. Suppose that the immediately prior LazyAMT, when computed, will add binding  $5 : a$ :



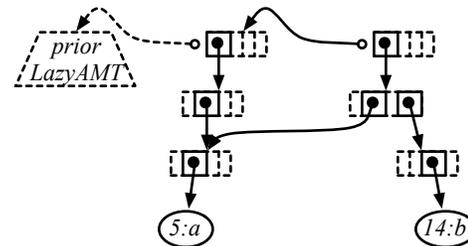
If we then look up key 5, or 000101 in binary, from the rightmost LazyAMT, we would first index 00 of the root node. We have already initialized this element from looking up key 14 before, so we simply walk to the next internal node and index 01. The element at 01 is uninitialized, but it is not along the lookup path of key 14, the key added to the rightmost LazyAMT. To continue the lookup of key 5, we retrieve the prior LazyAMT by forcing our lazy reference to it, causing it to be partially constructed:



Then, we initialize the element at 01 to point to the subtree under the element at the partial key 0001 in the prior LazyAMT, initializing the middle LazyAMT a bit more along the way:



We finish the lookup for key 5 as before, initializing the LazyAMTs a bit more:

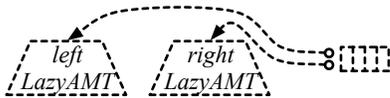


Note that we have simultaneously initialized parts of the rightmost LazyAMT and the middle LazyAMT because they share a common subtree; subsequent lookups of key 5 on either LazyAMT will become faster as a result.

**4.3.1.2 Merging two LazyAMTs:** The merge operation takes two LazyAMTs as input, which we call *left* and *right*, as well as a function,  $mergefn(lazy-opt-leftval, rightval)$ , that is called to merge values for the same key in both LazyAMTs. As the name of the arguments suggests,  $mergefn$  is called in an unusual, asymmetric manner in that it is called for all keys in the *right* LazyAMT, but not necessarily for all keys in the *left* LazyAMT. For each key in the *right* LazyAMT,  $mergefn$  is called with a lazy, optional value, representing the value for that key in the *left* LazyAMT, as the *lazy-opt-leftval* argument, and the value for the same key in the *right* LazyAMT as the *rightval* argument. This approach maximizes laziness in that we can compute a lazy value as soon as we determine a key exists in the one LazyAMT without immediately looking up the value for the same key in the other LazyAMT. For example,  $mergefn$  can be used to create a lazy linked-list by returning *lazy-opt-leftval* and *rightval* in a tuple.

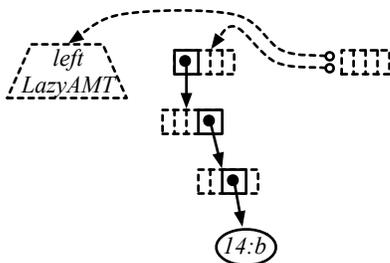
When we merge two LazyAMTs, we first create a new

root node of a new LazyAMT with two lazy references to the input LazyAMTs:

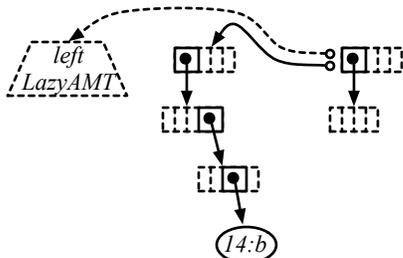


As before, the actual construction of the merged LazyAMT occurs when elements of internal nodes are initialized during lookups. We initialize an element in one of three ways, depending on whether the corresponding element (at the same partial lookup path) in the *right* LazyAMT points to an internal node, points to a leaf node, or is null. If the corresponding element points to an internal node, we create the next internal node and initialize the element to that node. If the corresponding element points to a leaf node, then we call *mergefn*, giving as *lazy-opt-leftval* a new lazy value that, when forced, looks up the same key in the *left* LazyAMT (returning null if the key does not exist), and as *rightval* the value at the leaf node. Otherwise, if the corresponding element is null, we initialize the element to point to the same subtree as the corresponding element in the *left* LazyAMT, or to be null if the corresponding element does not exist. Note that both the *left* and *right* LazyAMTs will be recursively initialized as necessary.

For example, suppose the *right* LazyAMT contains a single binding  $14 : b$ :

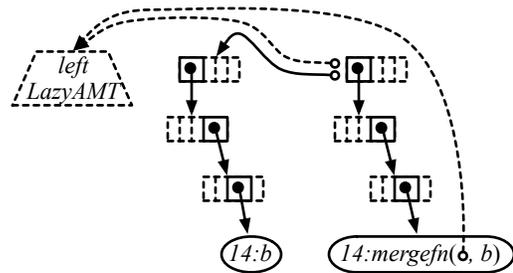


If we look up key 14, or 001110 in binary, we would first index element 00 of the root node. Since this is the first time we accessed this element, we initialize it by looking at the corresponding element in the *right* LazyAMT. The corresponding element points to an internal node, so we initialize the element to a new internal node:



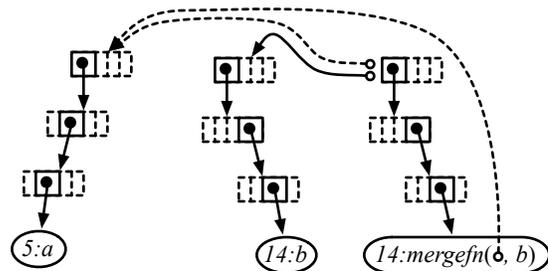
We repeat this process until we exhaust all bits in the lookup key and reach the corresponding leaf node. Because a binding exists for key 14 in the *right* LazyAMT, we initialize a new leaf node that binds key 14 to a merged value by first creating a new lazy value that

looks up the key 14 from the *left* LazyAMT, and calling *mergefn* on that lazy value as well as the value  $b$  from the *right* LazyAMT:

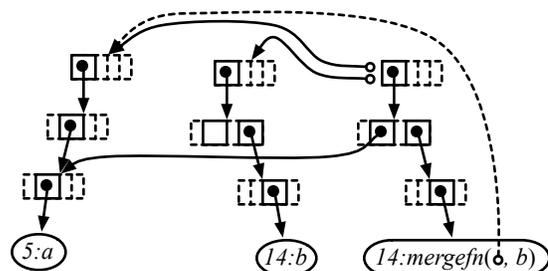


Note that we do not have to force the *left* LazyAMT to be computed immediately; the decision to force the *left* LazyAMT is deferred to *mergefn*. We finish the lookup by returning the newly merged value.

As another example, suppose the *left* LazyAMT contains a single binding  $5 : a$ :



If we look up key 5, or 000101 in binary, we would first index element 00 of the root node that is already initialized above. However, when we index element 01 of the next internal node, we would find that the corresponding element in the *right* LazyAMT is null. In this case, we look up the subtree under the corresponding element in the *left* LazyAMT, and initialize element 01 to it:



Note that we do not call *mergefn* in this case. We finish the lookup by walking into *left* LazyAMT to the leaf node for key 5, and returning  $a$ .

Finally, we represent the empty LazyAMT as a special instance that simply returns null for all key and partial key lookups, allowing us to avoid building a trie structure with no bindings.

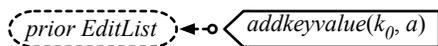
LazyAMT lookups are amortized  $O(1)$  time; the LazyAMT has a fixed depth of  $k/w$ , so each lookup

takes constant time to traverse the internal nodes to a leaf node. Similarly, adding a binding to a LazyAMT or merging two LazyAMTs take only amortized  $O(1)$  time and memory to create  $k/w$  internal nodes of size  $2^w$  each. The amortization of lookups, adding bindings, and merging LazyAMTs is due to laziness—most of the cost of adding a binding or merging two LazyAMTs is deferred to subsequent lookups. Adding the same key repeatedly would take an additional amortized  $O(1)$  memory each time (i.e., prior bindings are never removed, only shadowed). However, this is actually an advantage in EXPOSITOR as we are usually interested in how bindings to the same key change over time.

#### 4.3.2 EditList: Lazy Linked-List of Edit Operations

The second piece of the EditHAMT is the EditList, which is a lazy immutable set/map/multiset/multimap implemented as a lazy linked-list of edit operations.

When an operation such as add or remove is applied to an EditList, we simply append a new node to the EditList, labeling it with the given operation and arguments. For example, if we add a binding  $k_0 : a$  to an EditList (treating the EditList as a map or multimap), we create a new EditList node labeled  $addkeyvalue(k_0, a)$  with a lazy reference to the head of the prior EditList (i.e., the prior node):



We depict the EditList node as a box pointing to the left. Since all operations, including removals, are implemented by appending nodes, we do not need to know what the prior EditList contains, which we depict as a dotted arrow. And, as with the LazyAMT, we keep a lazy reference to the prior EditList which allows us to further delay any computation necessary to determine its contents, i.e., the prior EditList may not exist yet when we perform an addition or removal operation on it, which we depict as a dotted rounded box. Only when we first force the lazy reference to the prior EditList will we need to determine what kind of edit operation the prior EditList contains or if it is null, e.g., by making queries to UndoDB in EXPOSITOR.

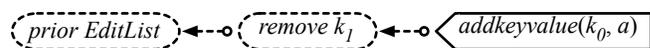
We support several different kinds of edit operations on EditLists, with the corresponding node labels:

- $add(k)$ : add element  $k$ , treating the EditList as a set or multiset;
- $addkeyvalue(k, v)$ : add a binding from key  $k$  to value  $v$ , treating the EditList as a map or multimap;
- $remove(k)$ : remove all elements/bindings for key  $k$ ;
- $removeone(k)$ : remove the latest element/binding for key  $k$ , treating the EditList as a multiset or multimap;
- $removekeyvalue(k, v)$ : remove the latest binding from key  $k$  to value  $v$ , treating the EditList as a multimap;
- $concat(lazy\_el)$ : concatenate the EditList  $lazy\_el$  as a lazy reference at this point.

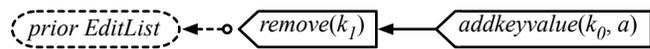
These are the building blocks for the corresponding EditHAMT operations listed in Fig. 4.

We implement EditList lookup in two different ways: one for set membership queries and map lookups, and another for multiset and multimap lookups.

**4.3.2.1 Set Membership Queries and Map Lookups:** We implement set membership queries and map lookups by traversing the EditList from head to tail (right to left), looking for the first node that contains an edit operation for the given key. For example, if we look up the value for key  $k_0$  in the above EditList, we would start by looking at the head node. Since the head node adds a binding from key  $k_0$ , we can finish the lookup and return value  $a$  without having to look at the prior EditList. As another example, suppose that the immediately prior node, when we force it to be computed, e.g., by making calls to UndoDB in EXPOSITOR, removes key  $k_1$ :



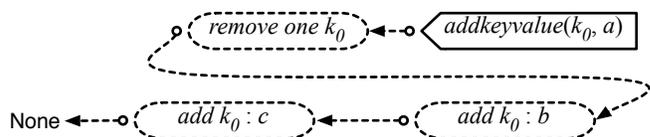
If we look up the value for key  $k_1$ , we would first look at the head node and skip it because it does not involve key  $k_1$ . Then, we would force the lazy reference to the prior node, causing it to be initialized if necessary, and look at it:



Here, we indicate that the tail has been forced with an unbroken arrow, and that the prior node has been initialized by replacing the dotted rounded box with pointed box. Since the prior node removes key  $k_1$ , we know that bindings no longer exist for key  $k_1$ , so we can finish the lookup and return null. This example shows that, once we have found a node that involves the lookup key, we no longer need to traverse the rest of the EditList. Also, because the reference to the prior EditList is lazy, the lookup process drives the construction of prior EditLists in an interleaved manner, just as in the LazyAMT.

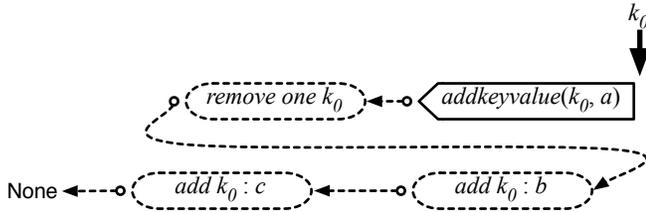
If we find a *concat* node during traversal, we handle that node by recursively looking up the concatenated EditList for the given key. If we do not find any relevant nodes in the concatenated EditList, we would then resume the lookup in the original EditList.

**4.3.2.2 Multiset and Multimap Lookups:** We implement multiset and multimap lookups lazily by returning a Python iterator that allows all values for a given key to be incrementally retrieved, typically via a `for x in values` loop. To illustrate how we implement multimap lookups, suppose we have the following EditList:



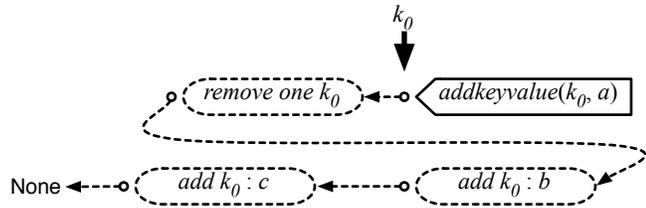
Reading backward from the tail, this EditList binds key  $k_0$  to value  $c$ , binds value  $b$  and removes it, then binds

value  $a$ ; i.e., it represents a map that contains bindings from  $k_0$  to values  $a$  and  $c$  but not  $b$ . A multimap lookup on this EditList for key  $k_0$  will return an iterator of all values bound to that key:



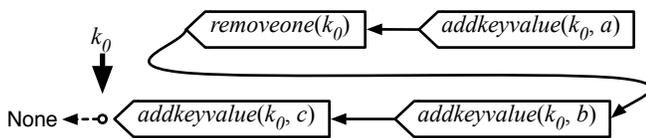
The iterator is associated with the key  $k_0$  and initially points to the beginning of the input EditList, which we depict as a large down-arrow labeled  $k_0$ .

The actual lookup does not begin until we retrieve a value from the iterator, which initiates a traversal of the input EditList to find a binding for key  $k_0$ :



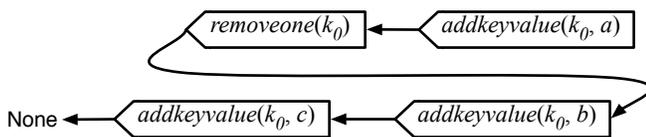
In this case, the head of the input EditList itself contains a binding to value  $a$ , so we return the value  $a$  and update the iterator to point to the tail of the input EditList (the head of the prior EditList), which we depict by moving the  $k_0$  down-arrow.

If we retrieve a value from the iterator again, the lookup continues from the head of the prior EditList:



The prior node removes the next binding for key  $k_0$ ; we temporarily note this as a pending removal. The node after that adds a binding for key  $k_0$ , but since we have a pending removal, we skip over that node. Next, we reach a node that binds key  $k_0$  to value  $c$ , so we return  $c$  and update the iterator as before.

If we retrieve a value from the iterator once more, the lookup will reach the end of the input EditList, which we depict as **None**, so we terminate the iterator. At this point, all nodes in the input EditList will have been initialized:



We implement multiset lookups identically by treating  $add(k)$  as if key  $k$  were bound to itself. Note in particular that, whereas a standard multiset lookup gives us the total number of values for a given key, a lazy multiset

lookup allows us to ask if there are at least  $n$  values for a given key. As we illustrated above, both multiset and multimap lookups are lazy in that the returned iterator will only initialize as many nodes of the input EditList as retrieved.

EditList set membership and map/multiset/multimap lookups are not particularly fast, since they take  $O(n)$ , where  $n$  is the number of edit operations in the EditList (i.e., the length of the EditList), to traverse the EditList to find a relevant binding. However, applying an edit operation takes only  $O(1)$  time and  $O(1)$  memory to create and append a single EditList node. Adding an element or binding to the same key repeatedly takes an additional  $O(1)$  memory each time, but as we explained for the LazyAMT, this is actually an advantage for EXPOSITOR as we are usually interested in how bindings change over time.

#### 4.3.3 EditList + Hash + LazyAMT = EditHAMT

As we described above, the LazyAMT provides fast amortized  $O(1)$  set/map lookups, but supports only fixed-width integer keys as well as addition and merging operations; it does not support removal operations or multiset/multimap lookups. Conversely, the EditList supports arbitrary keys, removal operations as well as multiset/multimap lookups, but lookups are a slow  $O(n)$  where  $n$  is the number of edit operations in the EditList. Both the LazyAMT and the EditList support lazy construction, i.e., we can perform operations such as addition or removal without knowing what the prior LazyAMT or EditList contains. Finally, each LazyAMT operation takes only an additional amortized  $O(1)$  time and memory over the prior LazyAMT, and likewise  $O(1)$  time and memory for the EditList.

We combine these two data structures to create the EditHAMT, a lazy data structure that is more capable than the LazyAMT and faster than the EditList. The key idea is build multiple EditLists, each containing only edits for keys with the same hash  $h$ , which we denote as  $editlist(h)$ , and use the LazyAMT to map hash  $h$  to  $editlist(h)$ . We can then look up a key  $k$  using the following steps:

- 1) compute the hash  $h$  of key  $k$ ;
- 2) look up  $editlist(h)$  from the LazyAMT of the EditHAMT;
- 3) look up key  $k$  from  $editlist(h)$ ;

The lookup process will cause the underlying LazyAMT or  $editlist(h)$  to be initialized as necessary.

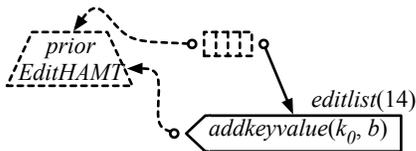
We construct EditHAMTs in one of two ways: we use the LazyAMT addition operation to perform addition or removal operations on an EditHAMT, and the LazyAMT merge operation to concatenate two EditHAMTs.

4.3.3.1 Performing Addition or Removal Operations on an EditHAMT: We take the following steps to perform an addition or removal operation for a given key  $k$  on an EditHAMT:

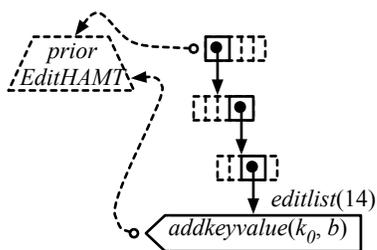
- 1) compute the hash  $h$  of key  $k$ ;
- 2) lazily look up the LazyAMT of the prior EditHAMT as  $lazyamt'$ ;
- 3) lazily look up  $editlist'(h)$  from  $lazyamt'$ , and append the given operation to it to create  $editlist(h)$ ;
- 4) add a new binding to  $lazyamt'$  from hash  $h$  to  $editlist(h)$  to create the updated EditHAMT;

where by lazily looking up we mean to create a lazy reference that, when forced, looks up  $lazyamt'$  or  $editlist'(h)$ , which allows the construction of the prior EditHAMT to also be lazy. Because both the LazyAMT and the EditList are lazy, the EditHAMT will be mostly uninitialized at first; (parts of) it will be initialized as necessary during lookup.

For example, suppose we add a binding from key  $k_0$  to value  $b$  to an EditHAMT, and key  $k_0$  has hash 14. We would create a new LazyAMT that maps hash 14 to a new  $editlist(14)$  that appends  $addkeyvalue(k_0, b)$  to the prior EditHAMT:

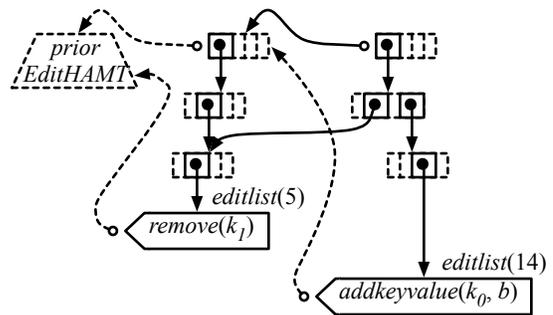


At first, most of the EditHAMT—the LazyAMT as well as the tail of the  $editlist(14)$ —is uninitialized; we indicate the uninitialized  $editlist(14)$  tail as a dotted arrow pointing to the prior EditHAMT. When we look up key  $k_0$ , we would look up hash 14 from the LazyAMT to find  $editlist(14)$  that we just added, then look up key  $k_0$  from  $editlist(14)$ . The head of  $editlist(14)$  adds key  $k_0$ , so we can return value  $b$  without looking at the tail. At the end of the lookup, the EditHAMT will be initialized as follows:



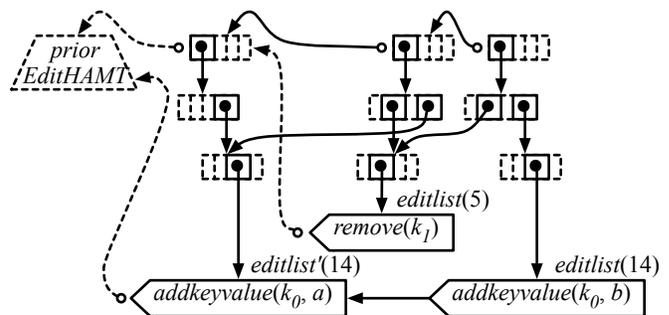
This example shows that we can apply an edit operation without knowledge of the prior EditHAMT, and since the key  $k_0$  was just added, we can look up key  $k_0$  without having to consult the prior EditHAMT. Both of these properties are inherited from the underlying LazyAMT and EditList.

We continue the example by considering the case of looking up a different key that was involved in an earlier edit operation. Suppose that the immediately prior EditHAMT removes key  $k_1$ , and key  $k_1$  has hash 5. When we look up key  $k_1$ , the EditHAMT will be initialized as follows:



Looking up the  $editlist(5)$  from the rightmost LazyAMT causes parts of the middle LazyAMT to be initialized and shared with the rightmost LazyAMT, and since the head of  $editlist(5)$  removes key  $k_1$ , we can return null without looking at its tail.

The two examples so far do not require traversing the tail of  $editlist(h)$ . We would need to do so if there was a hash collision or if we perform a multiset or multimap lookup. For example, suppose that an earlier EditHAMT added a binding from key  $k_0$  with hash 14 to value  $a$ , and we perform a multimap lookup of key  $k_0$  to find the second value. We have to initialize the tail of  $editlist(14)$  by looking up  $editlist'(14)$  from the prior EditHAMT. The immediately prior EditHAMT did not involve key  $k_0$  or hash 14; instead, we will partially initialize the earlier EditHAMT containing  $editlist'(14)$  and share it. Then, we retrieve  $editlist'(14)$  and initialize it as the tail of  $editlist(14)$ . At the end of the lookup, the EditHAMT will be initialized as depicted below:

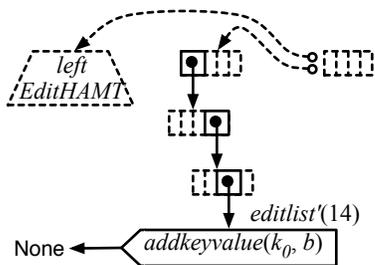


**4.3.3.2 Concatenating two EditHAMTs:** To concatenate two EditHAMTs, we call the LazyAMT merge operation with:

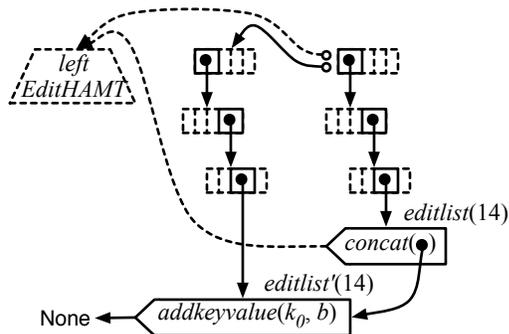
- the older EditHAMT as the *left* LazyAMT;
- the newer EditHAMT as the *right* LazyAMT;
- a *mergefn* function that creates a new  $editlist(h)$  by appending an EditList *concat* node containing  $editlist'(h)$  from the *right* EditHAMT to the lazy, optional value containing  $editlist''(h)$  from the *left* EditHAMT;

where the older and newer EditHAMTs are `lazy_ah1` and `lazy_ah2`, respectively, in Fig. 4.

For example, suppose we concatenate two EditHAMTs, where the *right* EditHAMT contains a single binding  $k_0 : b$  and key  $k_0$  has hash 14. We would create a new EditHAMT using the LazyAMT merge operation as described above:

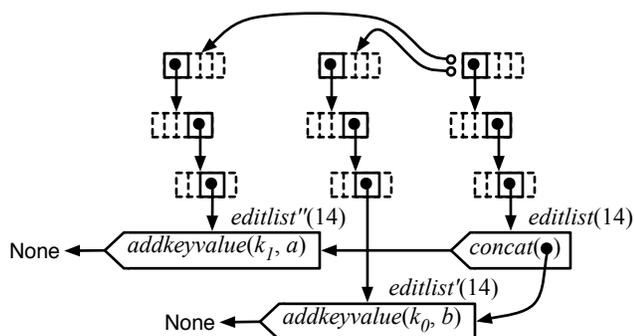


If we perform a map lookup on the concatenated EditHAMA to find the value for key  $k_0$ , we would look up the concatenated EditHAMA for hash 14 to retrieve  $editlist(14)$  and perform a map lookup on it. This will cause the concatenated EditHAMA to be initialized as follows:



The head of  $editlist(14)$  is a *concat* node created by the *mergefn* described above. When we look up key  $k_0$  from  $editlist(14)$ , we would recursively look up  $editlist'(14)$  from the *right* EditHAMA for the same key. Since the head of  $editlist'(14)$  adds key  $k_0$ , we finish the lookup by returning the mapped value  $b$ . We do not have to look at the tail of  $editlist(14)$  and can avoid forcing the *left* EditHAMA for now, which is a property inherited from the LazyAMT merge operation.

To consider an example that requires us to force the *left* EditHAMA, suppose that the *left* EditHAMA contains a single binding  $k_1 : a$  where key  $k_1$  also has hash 14. If we look up key  $k_1$  from the concatenated EditHAMA, it will be initialized in the following manner:



We would look up key  $k_1$  from  $editlist(14)$ , and recursively look up  $editlist'(14)$  for the same key. Because  $editlist'(14)$  does not contain key  $k_1$ , we would then resume the lookup in  $editlist(14)$ , forcing its tail to retrieve  $editlist''(14)$  from the *left* EditHAMA, and finally return

the mapped value  $a$ .

The combination of the LazyAMT and the EditList enables the EditHAMA to support all operations that the EditList supports, reduces the EditList lookup cost to amortized  $O(1)$  if we assume no hash collisions, and takes only an additional amortized  $O(1)$  time and memory for each edit operation. However, multiset and multimap lookups take amortized  $O(n)$  time where  $n$  is the number of *removeone* and *removekeyvalue* operations.

#### 4.4 Comparison with Other Data Structures

Compared to Python sets, which are implemented as hash tables, it is more memory efficient to make an updated copy of the EditHAMA, since only a constant number of nodes in the underlying LazyAMT are created, than it is to make a copy of the bucket array in the hash table underlying Python sets, which can be much larger. This makes it viable to store every intermediate EditHAMA as it is created in a trace, as each EditHAMA only requires an additional amortized  $O(1)$  memory over the prior EditHAMA. In our current implementation, a trace of EditHAMAs is cheaper than a trace of Python sets (which requires deep copying) if, on average, each EditHAMA or set in the trace has more than eight elements.

It is also common to implement sets or maps using self-balancing trees such as red-black trees or AVL trees. However, we observe that it is not possible to make these tree data structures as lazy as the EditHAMA. In these data structures, a rebalancing operation is usually performed during or after every addition or removal operation to ensure that every path in the tree does not exceed a certain bound. In particular, the root node may be swapped with another node in the process of rebalancing (in fact, every node may potentially be swapped due to rebalancing). This means that the root node of self-balancing trees is determined by the entire history of addition and removal operations. Thus, we would be forced to compute the entire history of addition and removal operations when we traverse the root node to look up a key, defeating laziness.

We also observe a similar issue arising with hash tables. Hash tables are based on a bucket array that is used to map hash values to keys. Typically, the bucket array is dynamically resized to accommodate the number of keys in the hash table and to reduce hash collisions. However, the number of keys in the hash table is determined by the entire history of addition and removal operations. As a result, we would be forced to compute the entire history of addition and removal operations before we can use the bucket array to map a hash value to a key, defeating laziness.

Furthermore, the EditHAMA suffers much less from hash collisions than hash tables. The LazyAMT in the EditHAMA is a *sparse* integer map, unlike the bucket array in hash tables, and thus can be made much larger

```

189 #define LOOP_I 16
190 #define LOOP_J 16
191 #define LOOP_K 16
192 #define LOOP_L 8
193 void foo(int x, int y) {}
194 void bar(int z) {}
195 int main(void) {
196     int i, j, k, l;
197     for (i = 0; i < LOOP_I; i++) {
198         for (j = 0; j < LOOP_J; j++) {
199             for (k = 0; k < LOOP_K; k++) {
200                 bar(i * LOOP_K + k);
201             }
202             foo(i, i * LOOP_J + j);
203             for (l = 0; l < LOOP_L; l++) {
204                 bar(i * LOOP_L + l);
205             }
206         }
207     }
208     return 0;
209 }

```

Fig. 6. Micro-benchmark test program.

while using little memory, which reduces the likelihood of hash collisions.

## 5 MICRO-BENCHMARKS

We ran two micro-benchmarks to evaluate the efficiency of EXPOSITOR. In the first micro-benchmark, we evaluated the advantage of laziness by comparing a script written in EXPOSITOR against several other equivalent scripts written using non-lazy methods. In the second micro-benchmark, we compared the performance of scripts using the EditHAMT against other equivalent scripts that use non-lazy data structures.

### 5.1 Test Program

For both micro-benchmarks, we use the test program in Fig. 6 as the subject of our EXPOSITOR scripts. This program consists of two do-nothing functions, `foo` on line 193 and `bar` on line 194, and the main function on lines 195–209 that calls `foo` and `bar` in several nested loops.

### 5.2 Experimental Setup

We run both micro-benchmarks on a 32-bit Ubuntu Linux 11.04 virtual machine (since UndoDB runs only on Linux), set up with 8 cores and 8 GB of RAM in VMware Fusion 4.1.4 on Mac OS X 10.6.8 running on a Mac Pro with two 2.26 GHz quad-core Intel Xeon processors and 16 GB of RAM. We use UndoDB version 3.5.1234, a developmental version of GDB (CVS revision as of October 4, 2012), and Python version 2.7.5.

### 5.3 Evaluating the Advantage of Trace Laziness

In our first micro-benchmark, we evaluate the advantage of trace laziness using the following procedure. We first start EXPOSITOR on the test program in Fig. 6, and run the following script:

```

210 foo_trace = the_execution.breakpoints("foo")
211 trace1 = foo_trace.filter(
212     lambda snap: int(snap.read_arg("x")) % 2 == 0)

```

This script creates a trace named `foo_trace` of calls to `foo`, and a trace named `trace1` that keeps only calls to `foo` where the argument `x` is even. We then measure the time it takes to call `get_after` to find the first item in `trace1` after time 0. We repeat this measurement to find the second item, the third item, and so forth, until there are no more items left in `trace1`. Next, we create another trace named `trace2`:

```

213 trace2 = foo_trace.filter(
214     lambda snap: int(snap.read_arg("x")) % 2 == 1)

```

This trace is similar to `trace1`, but keeps only calls to `foo` where `x` is odd. We then measure again the time it takes to call `get_after` to find each item in `trace2`. Finally, we restart EXPOSITOR and repeat the entire procedure, but use `get_before` to find all items in `trace1` and `trace2` starting from the end of the execution, instead of using `get_after`.

We compare the above script against several other equivalent scripts that are not lazy, written either in a variant of EXPOSITOR with trace laziness disabled, or using the standard GDB Python API with or without UndoDB. We disable trace laziness in EXPOSITOR by immediately performing the equivalent of `get_after(t)` on lazy nodes as they are created, where `t` is the beginning of the time interval on those nodes, and replacing the lazy nodes by the computed contents.

The results are shown in Fig. 7a for the procedure using `get_after`, and Fig. 7b for the procedure using `get_before`. The  $x$ -axes are labeled “action #”, and indicate particular actions that are taken during the benchmarking procedure:

- action 0 corresponds to creating `trace1`;
- actions 1–128 correspond to calling `get_after` or `get_before` repeatedly to get all items in `trace1`;
- action 129 (the vertical gray line) corresponds to creating `trace2`;
- actions 130–257 correspond to calling `get_after` or `get_before` repeatedly to get all items in `trace2`.

For scripts using the standard GDB Python API, action 0 and action 129 correspond instead to creating the equivalent of `trace1` or `trace2`, i.e., creating a standard Python list containing the times, rather than snapshots, of the relevant calls to `foo`. The  $y$ -axes indicate cumulative time in seconds, i.e., the total time it takes to perform all actions up to a particular action, and is mean-averaged over 31 runs.

The startup plot simply marks the time it takes for EXPOSITOR to call the GDB start command to start the execution, as well as to run any EXPOSITOR-specific startup initialization, before running any scripts. The `lazy_trace` plot corresponds to the script written in EXPOSITOR above. The `strict_trace` plot corresponds to the same script, but uses a variant of EXPOSITOR with trace laziness disabled. The `gdb_python` plot corresponds to

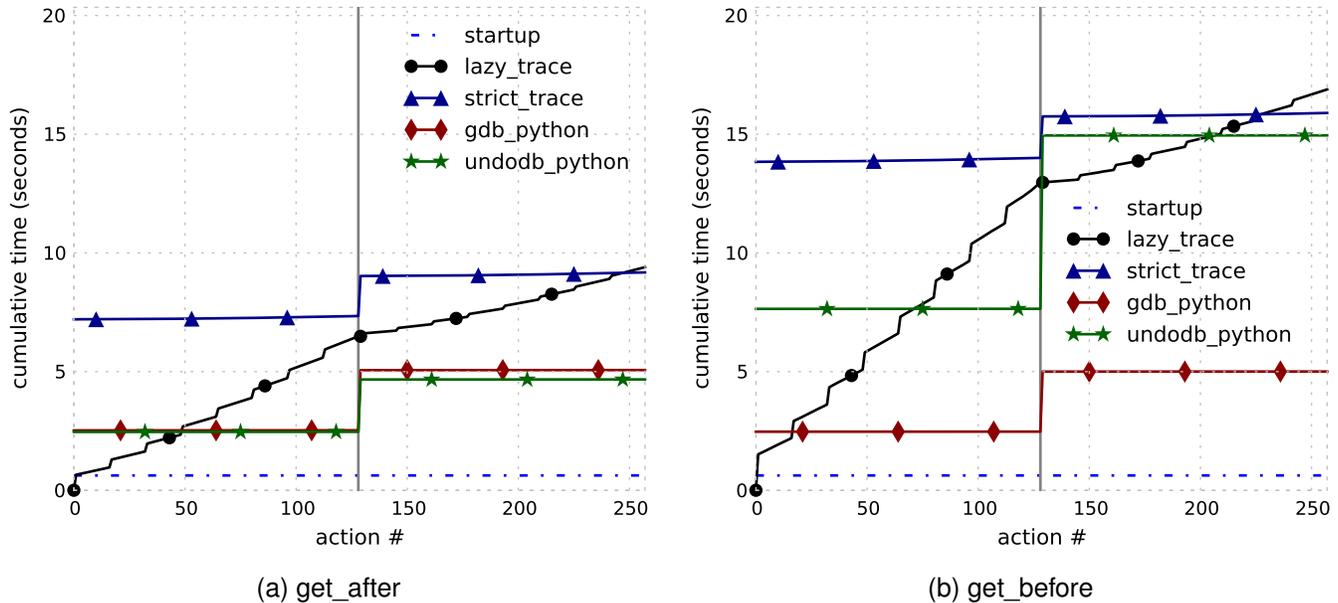


Fig. 7. The time it takes to get all items in two traces using lazy or non-lazy scripts.

a script written using the standard GDB Python API without the time-travel features of UndoDB, restarting the execution at action 129 (when the equivalent of trace2 is created), and without caching intermediate computation. Note that because `gdb_python` does not use time travel, the `gdb_python` scripts in Fig. 7a and Fig. 7b are the same, i.e., they both run the execution forward only, and `gdb_python` records times instead of snapshots; plotting `gdb_python` in Fig. 7b allows us to compare forward execution against backward execution. Lastly, the `undodb_python` plot uses a nearly identical script as `gdb_python`, but uses UndoDB to rewind the execution at action 129 instead of restarting the execution, and runs the execution backward in Fig. 7b.

From Fig. 7a, we can see that `lazy_trace` takes zero time to perform action 0, whereas all the other implementations take some non-zero amount of time. This is due to laziness—`lazy_trace` defers the startup cost until the first `get_after` call is made in action 1. We also note that `strict_trace` is slower than all other implementations, which suggests that the trace data structure has high overhead when laziness is disabled, and that `undodb_python` is slightly faster than `gdb_python` at action 129, since it is faster to rewind an execution than to restart it.

As we expect from laziness, each call to `get_after` in `lazy_trace` takes a small additional amount of time, whereas the other non-lazy implementations do not take any additional time (since all relevant calls to `foo` have already been found at action 0). When we have found about 40% items from trace1, the cumulative time of `lazy_trace` reaches that of `gdb_python`. This tells us that, as long as we do not make queries to more than 40% of an execution, it takes us less time to construct and query a lazy trace, compared to other non-lazy imple-

mentations. This is actually the common scenario in debugging, where we expect programmers to begin with some clues about when the bug occurs. For example, a stack corruption typically occurs near the end of the execution, so we would likely only have to examine the last few function calls in the execution.

Furthermore, we observe that the slope of `lazy_trace` is shallower at actions 130–257. This is because trace2 reuses `foo_trace` which was fully computed and cached during actions 1–128. Thus, EXPOSITOR does not have to perform as much work to compute trace2. `strict_trace` also benefits from caching since it uses a (non-lazy) variant of the trace data structure. In contrast, both `gdb_python` and `undodb_python` do not reuse any computation, so action 129 takes the same amount of time as action 0.

Fig. 7b shows the results of the benchmark procedure using `get_before`. We can see that it is much slower to use `get_before` than `get_after`—all scripts but `gdb_python` take longer than in Fig. 7a (`gdb_python` actually runs forward as we noted above). This is because these scripts have to run the execution in two passes: first to get to the end of the execution, then to execute the `get_before` calls back to the beginning of the execution. Unlike other scripts, the `gdb_python` script only has to run forward once and not backward, and so is much faster. Still, `lazy_trace` can be faster than `gdb_python`, if queries are made to fewer than about 10% of an execution.

We note that the results of this micro-benchmark actually suggest a lower bound to the advantage of trace laziness. This micro-benchmark is based on a very simple EXPOSITOR script that filters calls to `foo` using a simple predicate. Therefore, the time used for each script is dominated by the time it takes to set a breakpoint at `foo` and to run the execution, forward or backward, until the breakpoint. To a lesser extent, the trace data structure in

lazy\_trace adds overhead to the time used in comparison to gdb\_python. The filter predicate in lazy\_trace and the equivalent predicate in gdb\_python takes very little time in contrast. We expect more complex EXPOSITOR scripts to spend more time in programmer-provided helper functions such as the filter predicate or the scan operator, which will mask the overhead of the trace data structure.

#### 5.4 Evaluating the Advantage of the EditHAMT

In our second micro-benchmark, we evaluate the advantages of the EditHAMT data structure using the following procedure. We first create a trace of EditHAMTs:

```

215 bar_maps = the_execution.breakpoints("bar") \
216     .map(lambda snap: edithamt.addkeyvalue(
217         None, int(snap.read_arg("z")), snap)) \
218     .scan(edithamt.concat)

```

For each call to bar(z), we create a new EditHAMT that adds a binding from the argument z to the snapshot of that call. In other words, an EditHAMT in bar\_maps at time  $t$  contains bindings from arguments z to the corresponding bar(z) calls preceding and including the bar(z) call at time  $t$ .

We then create another trace that looks up values from bar\_maps:

```

219 bar_of_foos = the_execution.breakpoints("foo")
220     .trailing_merge(
221         lambda snap, bar_map:
222             bar_map.force().find(int(snap.read_arg("y"))),
223         bar_maps)

```

For each call to foo(x, y), we use the trailing\_merge method to look up the immediately prior EditHAMT in bar\_maps, and then look up that EditHAMT for the most recent bar(z) call where  $y = z$ .

Next, we measure the time it takes to call get\_after to look up the first item from bar\_of\_foos, which includes the time it takes to compute the EditHAMTs in bar\_maps as necessary, as well as to compute parts of the bar\_maps and bar\_of\_foos traces. We also measure the additional memory used after the call to get\_after by Python,<sup>5</sup> which includes the memory required to cache the intermediate computation of the EditHAMTs in bar\_maps as well as that of the bar\_of\_foos and bar\_maps traces, but does not include the memory usage of other parts of GDB as well as UndoDB. We repeat these measurements to find the second item, the third item, and so forth, until there are no more items in bar\_of\_foos. Finally, we restart EXPOSITOR and repeat the entire procedure using get\_before to find all items in bar\_of\_foos starting from the end of the execution, instead of using get\_after.

For this micro-benchmark, we set the key size of the EditHAMT to  $k = 35$  bits and its internal node index bit-width to  $w = 5$  bits, which gives it a maximum depth of 7. These parameters are suitable for 32-bit hash values that are padded to 35 bits. We compare the above script

5. We use Python's sys.getsizeof and gc.get\_objects functions to measure Python's memory usage.

against several other equivalent scripts using other data structures, as well as a script that uses the EditHAMT in a variant of EXPOSITOR with trace laziness disabled as described in Sec. 5.3, and a script that uses the standard GDB Python API without the time-travel features of UndoDB and a list of Python dicts (hash table) as the equivalent of bar\_maps.

The results are shown in Fig. 8: Fig. 8a and Fig. 8b show the time measurements using get\_after and get\_before, respectively, while Fig. 8c and Fig. 8d show the memory measurements using get\_after and get\_before, respectively. The  $x$ -axes are labeled "action #", and indicate particular actions that are taken during the benchmarking procedure:

- action 0 correspond to creating bar\_maps and bar\_of\_foos;
- actions 1–256 corresponds to calling get\_after or get\_before repeatedly to get all items in bar\_of\_foos.

For the script using the standard GDB Python API, action 0 correspond instead to creating a list of Python dicts mapping the arguments z of bar to times, rather than snapshots, of calls to bar. The  $y$ -axes indicate cumulative time in seconds or cumulative memory usage in bytes, i.e., the total time or memory it takes to perform all actions up to a particular action, and is mean-averaged over 31 runs.

The startup plot simply marks the time or memory it takes for EXPOSITOR to call the GDB start command to start the execution, as well as to run any EXPOSITOR-specific startup initialization, before running any scripts. The lazy\_trace\_edithamt plot corresponds to the EXPOSITOR script above that creates EditHAMTs, which are lazy, in bar\_maps. The lazy\_trace\_rbtrees plot corresponds to a similar script, but creates maps based on immutable red-black trees, which are not lazy, instead of EditHAMTs in bar\_maps. Likewise, the lazy\_trace\_python\_dict plot creates Python dicts, which are also not lazy, in bar\_maps. The strict\_trace\_edithamt plot corresponds to a script that uses the EditHAMT in a variant of EXPOSITOR with trace laziness disabled as described in Sec. 5.3. Lastly, the gdb\_python script corresponds to a script written using only the standard GDB Python API without the time-travel features of UndoDB. Note that because gdb\_python does not use time travel, the gdb\_python scripts in Figs. 8a–8d are all the same, i.e., they all run the execution forward only, and gdb\_python records times instead of snapshots; plotting gdb\_python in Fig. 8b and Fig. 8d allows us to compare forward execution against backward execution.

From Fig. 8a, we can see that the scripts that use EXPOSITOR, lazy\_trace\_X, take zero time to perform action 0, whereas strict\_trace\_edithamt and gdb\_python, which are not lazy, take some amount of time to do so, the former more than the latter. This is the result we expect based on the results in Sec. 5.3. Also, for lazy\_trace\_X, each get\_after calls takes a small additional amount of time. In particular, lazy\_trace\_edithamt takes less additional time than lazy\_trace\_rbtrees and lazy\_trace\_python\_dict. This

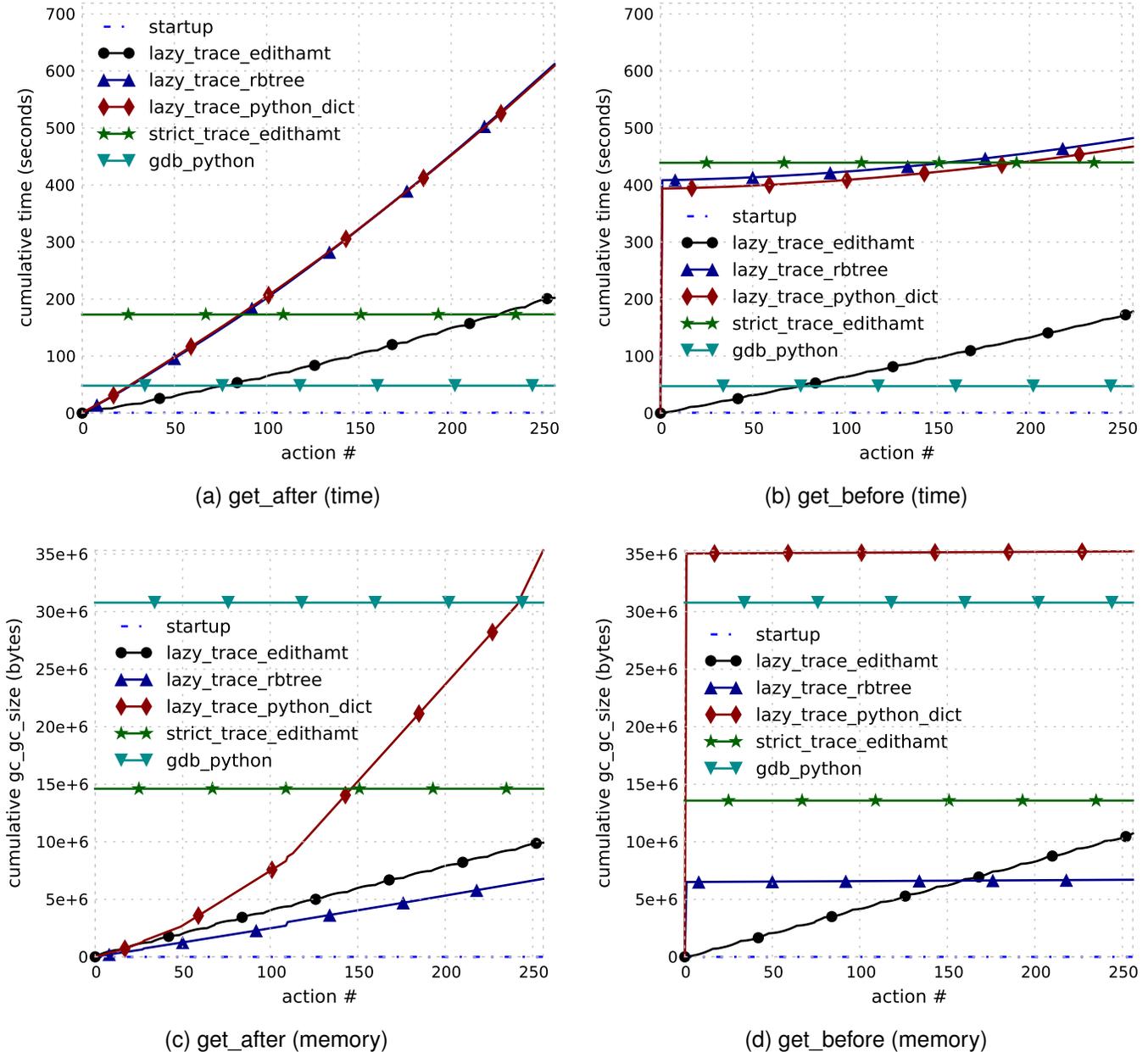


Fig. 8. The time and memory it takes to get all items in a trace computed by looking up items from an EditHAMT or other data structures.

is due to EditHAMT laziness: `lazy_trace_edithamt` only has to compute as many (parts) of the EditHAMTs in `bar_maps` as needed to answer the look up in `bar_of_foos`. In contrast, red-black trees and Python dicts are not lazy, so `lazy_trace_rbtree` and `lazy_trace_python_dict` have to compute all prior red-black trees or Python dicts, respectively, in `bar_maps`, even before answering the lookup in `bar_of_foos`. Also, since there are many more calls to `bar` than to `foo` in the test program (Sec. 5.1), and the `bar` call that matches a `foo` call occurs within a few calls to `bar`, `lazy_trace_edithamt` has to examine fewer `bar` calls than `lazy_trace_rbtree` or `lazy_trace_python_dict`. Furthermore, as long as we make fewer queries than about 30% of the items in `bar_of_foos`, it is faster to

use `lazy_trace_edithamt` than it is to use `gdb_python`. We also note that it is far easier to compose or reuse `lazy_trace_edithamt` than `gdb_python`. For example, if we later decide to compare the argument `x` of `foo` to the matching `bar` call, we can easily create a new trace that maps `foo` calls to their `x` argument and merge it with `bar_of_foos` in `lazy_trace_edithamt`, while we would need to modify and rerun `gdb_python` to collect the `x` arguments.

The results are quite different for `lazy_trace_rbtree` and `lazy_trace_python_dict` in Fig. 8b—action 0 still takes zero time, but action 1, the very first call to `get_before`, is very slow, contributing most of the cumulative time by the end of the micro-benchmark. This is because

that very first call to `get_before` retrieves the very last item in `bar_of_foos` near the end of the execution, which looks up one of the last red-black tree or Python dict in `bar_maps` to find the matching bar call. As we explained above, since red-black trees and Python dicts are not lazy, `lazy_trace_rbtrees` and `lazy_trace_python_dict` has to compute all prior red-black trees or Python dicts, respectively, and examine almost all bar calls when performing action 1. These cases highlight the importance of using lazy data structures in EXPOSITOR scripts—non-lazy data structures can completely defeat the advantage of trace laziness. Interestingly, `lazy_trace_edithamt` performs similarly whether we use `get_before` or `get_after`; this is because the computation of each item of `bar_of_foos` matches a `foo` call to a prior bar call by running the execution backwards, regardless of whether we use `get_before` or `get_after` to look up `bar_of_foos`. Note that we can easily change the computation of `bar_of_foos` to match a future bar call by using `rev_scan/rev_trailing_merge` instead of `scan/trailing_merge` in `lazy_trace_edithamt`, whereas making the same change in `gdb_python` would be significantly complicated by the lack of time-travel features.

Looking at memory usage, Fig. 8c shows that each `get_after` call in `lazy_trace_edithamt` and `lazy_trace_rbtrees` takes a small additional amount of memory. EditHAMTs use slightly more memory than red-black trees, despite fewer (parts of) EditHAMTs being computed due to laziness. However, we note that the memory cost of EditHAMTs is exaggerated in our Python-based implementation, since it makes extensive use of closures which are rather costly memory-wise in Python.<sup>6</sup> We believe that an implementation of EditHAMTs in more efficient languages such as C or OCaml would use much less memory. In contrast, `lazy_trace_python_set` is quite expensive, using an increasing amount of memory after each `get_after` call. This is because Python dicts are implemented using hash tables, and since each dict in `bar_maps` has to be distinct, we have to make a deep copy of the dicts. This eventually takes  $O(n^2)$  memory where  $n$  is the number of bar calls in the execution, which is confirmed by Fig. 8c. The `gdb_python` script uses almost as much memory as `lazy_trace_python_dict` by the end of the micro-benchmark, since they both create many Python dicts, except that `gdb_python` stores them in a Python list which has lower overhead than a trace. The `strict_trace_edithamt` script also uses a lot memory, more than `lazy_trace_edithamt` by the end of the micro-benchmark, since `strict_trace_edithamt` has to create every EditHamt in `bar_maps` (the EditHAMTs themselves are lazy), unlike `lazy_trace_edithamt` which only creates EditHAMTs in `bar_maps` when they are looked up.

We see a similar pattern in Fig. 8d for memory usage as in Fig. 8b for time usage: action 1 of `lazy_trace_rbtrees` and `lazy_trace_python_dict` contributes almost all of the cumulative memory usage by the end of the execution.

6. In 32-bit Python 2.7.1, a closure over a single variable takes 108 bytes, and each additional variable takes 28 bytes.

As we explained above for Fig. 8b, this is due to having to compute almost all red-black trees or Python dicts in `bar_maps`.

## 6 FIREFOX CASE STUDY: DELAYED DEALLOCATION BUG

To put EXPOSITOR to the test, we used it to track down a subtle bug in Firefox that caused it to use more memory than expected [11]. The bug report contains a test page that, when scrolled, creates a large number of temporary JavaScript objects that should be immediately garbage collected. However, in a version of Firefox that exhibits the bug (revision `c5e3c81d35ba`), the memory usage increases by 70MB (as reported by top), and only decreases 20 seconds after a second scroll. As it turns out, this bug has never been directly fixed—the actual cause is a data race, but the official fix instead papers over the problem by adding another GC trigger.

Our initial hypothesis for this bug is that there is a problem in the JavaScript garbage collector (GC). To test this hypothesis, we first run Firefox under EXPOSITOR, load the test page, and scroll it twice, temporarily interrupting the execution to call `the_execution.get_time()` just before each scroll, time  $t_{scroll1}$  and time  $t_{scroll2}$ , and after the memory usage decreases,  $t_{end}$ . Then, we create several traces to help us understand the GC and track down the bug, as summarized in Fig. 9.

We observe the GC behavior using a trace of the calls to `(gc_call)` and returns from `(gc_return)` function `js_GC` (Fig. 9a).<sup>7</sup> Also, we find out when memory is allocated or released to the operating system using `mmap2` and `munmap` traces of the same-named system calls (Fig. 9b). Printing these traces reveals some oddly inconsistent behavior: the GC is called only once after  $t_{scroll1}$ , but five times after  $t_{scroll2}$ ; and memory is allocated after  $t_{scroll1}$  and deallocated just before  $t_{end}$ . To make sense of these inconsistencies, we inspect the call stack of each snapshot in `gc_call` and discover that the first `js_GC` call immediately after a scroll is triggered by a scroll event, but subsequent calls are triggered by a timer.

We now suspect that the first scroll somehow failed to trigger the creation of subsequent GC timers. To understand how these timers are created, we write a function called `set_tracing` that creates a trace for analyzing set-like behavior, using EditHAMTs to track when values are inserted or removed, and apply `set_tracing` to create `timer_trace` by treating timer creation as set insertion, and timer triggering as set removal (Fig. 9c). This trace reveals that each `js_GC` call creates a GC timer (between `gc_call` and `gc_return` snapshots), except the `js_GC` call after the first scroll (and the last `js_GC` call because GC is complete).

To find out why the first `js_GC` call does not create a GC timer, we inspect call stacks again and learn that a GC timer is only created when the variable

7. The `index=-1` optional argument to `execution.breakpoints` indicates that the breakpoint should be set at the end of the function.

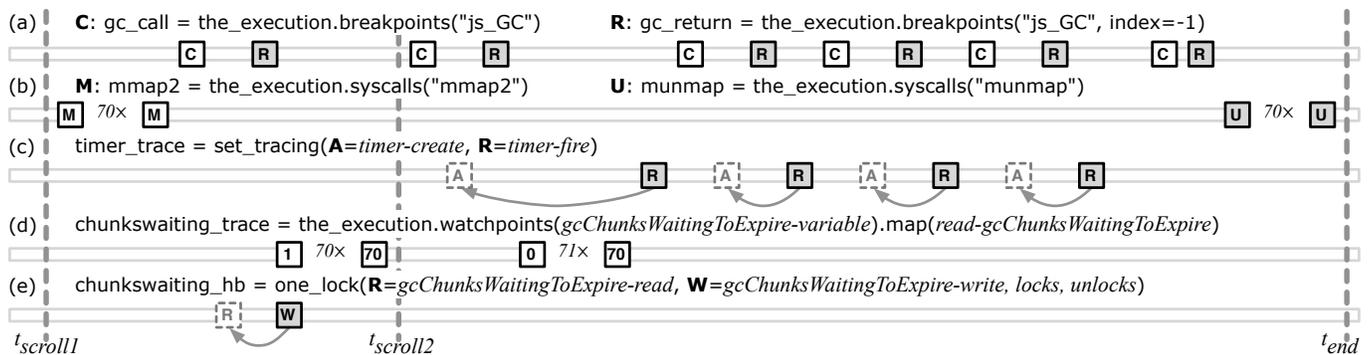


Fig. 9. Timeline of items in traces used to debug Firefox.

`gcChunksWaitingToExpire` is nonzero, and yet it is zero when the first `js_GC` returns (at the first `gc_return` snapshot). Following this clue, we create a watchpoint trace on `gcChunksWaitingToExpire` and discover that it remained zero through the first `js_GC` call and becomes nonzero only after the first `js_GC` returns. It stayed nonzero through the second scroll and second `js_GC` call, causing the first GC timer to be created after that (Fig. 9d).

We posit that, for the GC to behave correctly, `gcChunksWaitingToExpire` should become nonzero at some point during the first `js_GC` call. Inspecting call stacks again, we find that `gcChunksWaitingToExpire` is changed in a separate helper thread, and that, while the GC owns a mutex lock, it is not used consistently around `gcChunksWaitingToExpire`. This leads us to suspect that there is a data race. Thus, we develop a simple race detection script, `one_lock`, that works by comparing each access to a chosen variable against prior accesses from different threads (Sec. 4.1 explains how we track prior accesses), and checking if a particular lock was acquired or released prior to those accesses. For each pair of accesses, if at least one access is a write, and the lock was not held in one or both accesses, then there is a race, which we indicate as an item containing the snapshot of the prior access. We apply this race detector to `gcChunksWaitingToExpire` and confirm our suspicion that, after  $t_{scroll1}$ , there is a write that races with a prior read during the first `js_GC` call when the timer should have been created (Fig. 9e).

To give a sense of EXPOSITOR’s performance, it takes 2m6s to run the test page to  $t_{scroll2}$  while printing the `gc_call` trace, with 383MB maximum resident memory (including GDB, since EXPOSITOR extends GDB’s Python environment). The equivalent task in GDB/UndoDB without EXPOSITOR takes 2m19s and uses 351MB of memory (some difference is inevitable as the test requires user input, and Firefox has many sources of nondeterminism). As another data point, finding the race after  $t_{scroll1}$  takes 37s and another 5.4MB of memory.

The two analyses we developed, `set_tracing` and `one_lock`, take only 10 and 40 lines of code to implement, respectively, and both can be reused in other debugging contexts.

## 7 RELATED WORK

EXPOSITOR provides scripting for time-travel debuggers, with the central idea that a target program’s execution can be manipulated (i.e., queried and computed over) as a first-class object. Prior work on time-travel debugging has largely provided low-level access to the underlying execution without consideration for scripting. Of the prior work on scriptable debugging, EXPOSITOR is most similar to work that views the program as an event generator—with events seeded from function calls, memory reads/writes, etc.—and debugging scripts as database-style queries over event streams or as dataflow-oriented stream transformers. None of this scripting work includes the notion of time travel.

### 7.1 Time-Travel Debuggers

Broadly speaking, there are two classes of time-travel debuggers. *Omniscient debuggers* work by logging the state of the program being debugged after every instruction, and then reconstructing the state from the log on demand. Some examples of omniscient debuggers include ODB [12], Amber (also known as Chronicle) [6], Tralfamadore [13], and TOD [14]. In contrast, *replay debuggers* work by logging the results of system calls the program makes (as well as other sources of nondeterminism) and making intermediate checkpoints, so that the debugger can reconstruct a requested program state by starting at a checkpoint and replaying the program with the logged system calls. Several recent debuggers of this style include URDB [15] and UndoDB [5] (which we used in our prototype) for user-level programs, and TTVM [16] and VMware ReTrace [17] for entire virtual machines. EXPOSITOR could target either style of debugger in principle, but replay debugging scales much better (e.g., about  $1.7\times$  recording overhead for UndoDB vs.  $300\times$  for Amber). Engblom [18] provides a more comprehensive survey on time-travel debugging techniques and implementations.

The above work focuses on implementing time travel efficiently; most systems provide very simple APIs for accessing the underlying execution, and do not consider how time travel might best be exploited by debugging scripts.

Similarly, GDB’s Python environment simply allows a Python program to execute GDB (and UndoDB) commands in a callback-oriented, imperative style. This is quite tedious, e.g., just counting the number of calls to a particular function takes 16 lines of code (Sec. 2.2.1), and cannot be composed with other scripts (e.g., to refine the count to calls that satisfy predicate  $p$ ). EXPOSITOR’s notion of traces is simpler and more composable: function call counting can be done in one or two lines by computing the length of a breakpoint trace; to refine the count, we simply filter the trace with  $p$  before counting (Sec. 2.2).

Tralfamadore [19] considers generalizing standard debugging commands to entire executions, but does not provide a way to customize these commands with scripts.

Whyline is a kind of omniscient debugger with which users can ask “*why did*” and “*why didn’t*” questions about the control- and data-flow in the execution, e.g., “*why did this Button’s visible = true*” or “*why didn’t Window appear*” [20]. Whyline records execution events (adding  $1.7\times$  to  $8.5\times$  overhead), and when debugging begins, it uses program slicing [21] to generate questions and the corresponding answers (imposing up to a  $20\times$  further slowdown). Whyline is good at what it does, but its lack of scriptability limits its reach; it is hard to see how we might have used it to debug the Firefox memory leak, for example. In concept, Whyline can be implemented on top of EXPOSITOR, but limitations of GDB and UndoDB (in particular, the high cost of software watchpoints, and the inability to track data-flow through registers) makes it prohibitively expensive to track fine-grained data-flow in an execution. We plan to overcome this limitation in future work, e.g., using EDDI [22] to implement fast software watchpoints.

## 7.2 High-Level (Non-callback Oriented) Debugging Scripts

EXPOSITOR’s design was inspired by MzTake [3], a Scheme-based, interactive, scriptable debugger for Java based on *functional reactive programming* [4]. In MzTake, the program being debugged is treated as a source of *event streams* consisting of events such as function calls or value changes. Event streams can be manipulated with combinators that filter, map, fold, or merge events to derive new event streams. As such, an event stream in MzTake is like a trace in EXPOSITOR. Computations in MzTake are implicitly over the most recent value of a stream and are evaluated eagerly as the target program runs. To illustrate, consider our example of maintaining a shadow stack from Sec. 2.3. In MzTake, when the target program calls a function, a new snapshot event  $s$  becomes available on the calls stream. The `calls_rets` stream’s most recent event is the most recent of calls and rets, so MzTake updates it to  $s$ . Since `shadow_stacks` is derived from `calls_rets`, MzTake updates its most recent event by executing `map(int, s.read_retaddr())`.

This eager updating of event streams, as the program executes, can be less efficient than using EXPOSITOR. In particular, EXPOSITOR evaluates traces lazily so that computation can be narrowed to a few slices of time. In Sec. 2.3, we find the *latest* smashed stack address without having to maintain the shadow stack for the entire program execution, as would be required for MzTake. Also, EXPOSITOR traces are time indexed, but MzTake event streams are not: there is no analogue to `tr.get_at(i)` or `tr.slice(t0, t1)` in MzTake. We find time indexing to be very useful for interactivity: we can run scripts to identify an interesting moment in the execution, then explore the execution before and after that time. Similarly, we can learn something useful from the end of the execution (e.g., the address of a memory address that is double-freed), and then use it in a script on an earlier part of the execution (e.g., looking for where that address was first freed). MzTake requires a rerun of the program, which can be a problem if nondeterminism affects the relevant computation.

Dalek [23] and Event Based Behavioral Abstraction (EBBA) [24] bear some resemblance to MzTake and suffer the same drawbacks, but are much lower-level, e.g., the programmer is responsible for manually managing the firing and suppression of events. Coca [25] is a Prolog-based query language that allows users to write predicates over program states; program execution is driven by Prolog backtracking, e.g., to find the next state to match the predicate. Coca provides a *retrace* primitive that restarts the entire execution to match against new predicates. This is not true time travel but re-execution, and thus suffers the same problems as MzTake.

PTQL [26], PQL [27], and UFO [28] are declarative languages for querying program executions, as a debugging aid. Queries are implemented by instrumenting the program to gather the relevant data. In principle, these languages are subsumed by EXPOSITOR, as it is straightforward to compile queries to traces. Running queries in EXPOSITOR would allow programmers to combine results from multiple queries, execute queries lazily, and avoid having to recompile (and potentially perturb the execution of) the program for each query. On the other hand, it remains to be seen whether EXPOSITOR traces would be as efficient as using instrumentation.

## 8 CONCLUSION

We have introduced EXPOSITOR, a novel scriptable, time-travel debugging system. EXPOSITOR allows programmers to project a program execution onto immutable traces, which support a range of powerful combinators including map, filter, merge, and scan. The trace abstraction gives programmers a global view of the program, and is easy to compose and reuse, providing a convenient way to correlate and understand events across the execution timeline. For efficiency, EXPOSITOR traces are implemented using a lazy, interval-tree-like data structure. EXPOSITOR materializes the tree nodes on demand, ultimately calling UndoDB to retrieve appropriate

snapshots of the program execution. EXPOSITOR also includes the EditHMT, which lets script writers create lazy sets, maps, multisets, and multimaps that integrate with traces without compromising their laziness. We ran two micro-benchmarks that show that EXPOSITOR scripts using lazy traces can be faster than the equivalent non-lazy scripts in common debugging scenarios, and that the EditHMT is crucial to ensure that trace laziness is not compromised. We used EXPOSITOR to find a buffer overflow in a small program, and to diagnose a very complex, subtle bug in Firefox. We believe that EXPOSITOR holds promise for helping programmers better understand complex bugs in large software systems.

## ACKNOWLEDGMENTS

This research was supported by in part by National Science Foundation grants CCF-0910530 and CCF-0915978. We also thank Vic Zandy and Robert O'Callahan for helpful comments and inspiration to pursue this work, and Greg Law for providing prompt support and bug fixes for UndoDB.

## REFERENCES

- [1] R. O'Callahan. (2010) LFX2010: a browser developer's wish list. Mozilla. At 27:15. [Online]. Available: <http://vimeo.com/groups/lfx/videos/12471856#t=27m15s>
- [2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2006.
- [3] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss, "The design and implementation of a dataflow language for scriptable debugging," *Automated Software Engineering*, vol. 14, no. 1, pp. 59–86, Mar. 2007. doi:10.1007/s10515-006-0003-z
- [4] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '97. New York, NY, USA: ACM, 1997, pp. 263–273. doi:10.1145/258948.258973
- [5] Undo Software. What is UndoDB? Undo Software. [Online]. Available: <http://undo-software.com/product/undodb-overview> (accessed June 2013)
- [6] R. O'Callahan, "Efficient collection and storage of indexed program traces," 2006. [Online]. Available: <http://www.ocallahan.org/Amber.pdf> (accessed June 2013)
- [7] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 365, 1996. [Online]. Available: <http://www.phrack.org/issues.html?issue=49&id=14#article> (accessed June 2013)
- [8] J. D. Blackstone. Tiny HTTPd. [Online]. Available: <http://tinyhttpd.sourceforge.net> (accessed June 2013)
- [9] S. Designer, "'return-to-libc' attack," *Bugtraq*, Aug. 1997. [Online]. Available: <http://seclists.org/bugtraq/1997/Aug/63> (accessed June 2013)
- [10] P. Bagwell, "Ideal hash trees," *École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Tech. Rep.*, 2001. [Online]. Available: <http://infoscience.epfl.ch/record/64398>
- [11] A. Zakai. (2011, May) Bug 654028 - 70mb of collectible garbage not cleaned up. Bugzilla@Mozilla. [Online]. Available: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=654028](https://bugzilla.mozilla.org/show_bug.cgi?id=654028) (accessed June 2013)
- [12] B. Lewis, "Debugging backwards in time," in *Proceedings of the Fifth International Workshop on Automated Debugging*, ser. AADEBUG '03, 2003. [Online]. Available: <http://arXiv.org/abs/cs/0310016>
- [13] G. Lefebvre, B. Cully, C. Head, M. Spear, N. Hutchinson, M. Feeley, and A. Warfield, "Execution mining," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 145–158. doi:10.1145/2151024.2151044
- [14] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 535–552. doi:10.1145/1297027.1297067
- [15] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston, "URDB: a universal reversible debugger based on decomposing debugging histories," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, ser. PLOS '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:5. doi:10.1145/2039239.2039251
- [16] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 1–1. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247360.1247361>
- [17] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman, "ReTrace: collecting execution trace with virtual machine deterministic replay," in *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*, ser. MoBS '07, 2007. [Online]. Available: <http://www-mount.ece.umn.edu/~jjyi/MoBS/2007/program/01C-Xu.pdf>
- [18] J. Engblom, "A review of reverse debugging," in *System, Software, SoC and Silicon Debug Conference*, ser. S4D '12. Piscataway, NJ, USA: IEEE, 2012, pp. 1–6. [Online]. Available: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6338149>
- [19] C. C. D. Head, G. Lefebvre, M. Spear, N. Taylor, and A. Warfield, "Debugging through time with the Tralfamadore debugger," in *Runtime Environments, Systems, Layering and Virtualized Environments*, ser. RESOLVE '12, 2012. [Online]. Available: [http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session4\\_paper1.pdf](http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session4_paper1.pdf)
- [20] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 301–310. doi:10.1145/1368088.1368130
- [21] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005. doi:10.1145/1050849.1050865
- [22] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: efficient debugging using dynamic instrumentation," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC '08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 147–162. doi:10.1007/978-3-540-78791-4\_10
- [23] R. A. Olsson, R. H. Crawford, and W. W. Ho, "A dataflow approach to event-based debugging," *Software: Practice and Experience*, vol. 21, no. 2, pp. 209–229, Feb. 1991. doi:10.1002/spe.4380210207
- [24] P. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, ser. PADD '88. New York, NY, USA: ACM, 1988, pp. 11–22. doi:10.1145/68210.69217
- [25] M. Ducassé, "Coca: an automated debugger for C," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 504–513. doi:10.1145/302405.302682
- [26] S. F. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 385–402. doi:10.1145/1094811.1094841
- [27] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 365–383. doi:10.1145/1094811.1094840
- [28] M. Auguston, C. Jeffery, and S. Underwood, "A framework for automatic debugging," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ser. ASE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 217–222. doi:10.1109/ASE.2002.1115015