TRAVIOLI: A Dynamic Analysis for Detecting Data-Structure Traversals

Rohan Padhye and Koushik Sen
EECS Department
University of California, Berkeley, USA
{rohanpadhye, ksen}@cs.berkeley.edu

***Abstract*—Traversal is one of the most fundamental operations on data structures, in which an algorithm systematically visits some or all of the data items of a data structure. We propose a dynamic analysis technique, called TRAVIOLI, for detecting data-structure traversals. We introduce the concept of acyclic execution contexts, which enables precise detection of traversals of arrays and linked data structures such as lists and trees in the presence of both loops and recursion. We describe how the information reported by TRAVIOLI can be used for visualizing data-structure traversals, manually generating performance regression tests, and for discovering performance bugs caused by redundant traversals. We evaluate TRAVIOLI on five real-world JavaScript programs. In our experiments, TRAVIOLI produced fewer than 4% false positives. We were able to construct performance tests for 93.75% of the reported true traversals. TRAVIOLI also found two asymptotic performance bugs in widely used JavaScript frameworks D3 and express.**

I. INTRODUCTION

Data structures form the building blocks of almost all programs. As programs grow large, the implementations of their data structures and of the code modules that use these data structures also become complex. Researchers have developed several tools to identify, to visualize, and to reason about data structures using static or dynamic program analysis techniques [10, 16, 21, 24, 28, 31]. Most of these tools focus on discovering a concise *representation* or *abstraction* of data structures in program memory at one or more program-locations during program execution.

We focus on understanding where and how data structures are *traversed* in a program. Traversal is perhaps one of the most fundamental operations on data structures, in which an algorithm systematically visits some or all of the data items of a data structure [29]. The running time of program functions that perform traversals usually increases with the increase in the size of the input data structures. Thus, a proper understanding of how a program traverses its data structures is crucial for characterizing the program's performance.

We propose a dynamic analysis technique, called TRAVIOLI, to detect data-structure traversals in a program. The technique works by analyzing read-events generated by an execution of the program. TRAVIOLI reports a data-structure traversal if it finds that 1) the program reads distinct memory-locations at a single program-location, and 2) those memory-locations either belong to the same object, or belong to different objects connected by a series of pointers. A key contribution of this work is the notion of *acyclic execution contexts* (AECs), which enables precise detection of traversals in the presence of loops and recursion with very few false positives.

We describe three applications of TRAVIOLI and AECs. First, we show how we can construct an *access graph* to help visualize data-structure traversals found in a program. Second, we show how we can use TRAVIOLI to aid in manually constructing performance regression tests. Finally, we show how TRAVIOLI can be used to detect redundant-traversal bugs.

A key advantage of TRAVIOLI is that it can detect a traversal even if a program is executed on a small unit test—the program does not need to execute a program-location many times to detect a traversal. Another key advantage of TRAVIOLI is that it can detect a traversal even if the traversal involves recursive function calls and loop iterations.

We have implemented TRAVIOLI for JavaScript and made it publicly available at https://github.com/rohanpadhye/travioli. We applied TRAVIOLI to 5 popular JavaScript projects. In our experimental evaluation, we found that TRAVIOLI reported false positives only 4% of the time. In 93.75% of reported true traversals, we managed to create a performance regression test. TRAVIOLI discovered two previously unknown performance bugs due to redundant traversals in widely used JavaScript frameworks—`D3` and `express`, which have been confirmed by the respective project's developers.

II. OVERVIEW

We propose a technique to identify functions that traverse input data structures and whose running time could be arbitrarily increased by increasing the size of the input data structure. We call such functions *traversing functions*. In the rest of the section, we present a series of examples written in JavaScript to motivate the definition of a traversing function and informally describe our approach for identifying such functions using dynamic program analysis. Section III formalizes these ideas concretely.

A. Traversing Functions

Consider the function `sum` in Figure 1. The function iterates over an input array of objects, `arr`, and computes the sum of the `val` field of the objects it contains. The function is an example of a simple data-structure traversing function. The running time of the function can be increased by increasing the size of the input array.

```
1 /* Sum values in records in array */
2 function sum(arr) {
3   var result = 0, record, i;
4   for(i = 0; i < arr.length; i++){
5     record = arr[i];
6     result += record.val;
7   }
8   return result;
9 }
```

Fig. 1. A function that traverses an array.

```
1 /* Compute the length of linked list  */
2 function len(list) {
3   var count = 0;
4   while (list != null) {
5     count++;
6     list = list.next;
7   }
8   return count;
9 }
```

Fig. 2. A function that traverses a linked list.

Although in this example we could easily identify the input to the function (i.e. the array `arr`), this may be non-trivial for complex functions where inputs could be passed via global or static variables. We define *read-footprint* to precisely capture the set of inputs to a function. A *memory-location* is the address of a piece of memory which stores a program value that can be read by a program. A memory-location is often denoted in a program by a variable, an element of an array, or a field of an object. The *read-footprint* of a function consists of all memory-locations that are read by an execution of the function without any prior write to them by the execution. Such memory-locations could be treated as the input to the function. For example, the read-footprint of the `sum` function consists of the array `arr`, all its elements (accessed via `arr[i]`), the `length` field of the array (accessed via `arr.length`), and the field `val` of the objects stored in the array (accessed via `record.val`). In contrast, the memory-locations denoted by the variables `i`, `record` and `result` are not part of the read-footprint, because, in any execution of `sum`, `sum` first writes them before reading them.

Given the definition of a read-footprint, we can define a traversing function as follows: we say that a function is a *traversing function* if the size of its read-footprint can be arbitrarily increased by providing suitable inputs, possibly of larger size. If a function is a traversing function, then we say that the function contains a *traversal*.

The function `sum` in Figure 1 contains a traversal because the size of the read-footprint increases if the size of the input array `arr` is increased.

The function `len` in Figure 2 is another example of a traversing function. The read-footprint of the `len` function consists of the memory-location denoted by `list` and the memory-locations denoted by the `next` field of all objects reachable from `list` by following the `next` field zero or more times. The read-footprint of this function can be increased by increasing the size of the list passed as an argument.

In contrast, the function `addPair` in Figure 3 is not a traversing function. The function `addPair` adds the values of the first two elements of the input array. While this

```
1 /* Add values from a pair of array elements. */
2 function addPair(arr) {
3   var rx = arr[0];
4   var ry = arr[1];
5   return rx.val
6        + ry.val;
7 }
```

Fig. 3. A non-traversing function.

function also reads multiple elements of `arr`, it is not a traversing function because the size of its read-footprint is always bounded regardless of the size of the input array or the values it contains.

B. Detecting Traversing Functions

The problem of determining if a function contains a traversal is undecidable in general. However, in many cases, one can determine whether a function has a traversal either by analyzing the source code or by analyzing an execution of the function. We propose a dynamic analysis technique, called TRAVIOLI, to determine if a function contains a traversal. TRAVIOLI works by checking a set of conditions on an execution of the function—if the conditions are satisfied then we say that the function contains a *possible* traversal. Our technique is approximate in the sense that it can give both false positives and negatives. However, we have identified a set of conditions which, if satisfied, often accurately indicate the presence of a traversal. A key feature of TRAVIOLI is that we do not need to invoke the function on an input having a large read-footprint—TRAVIOLI can detect a traversal by analyzing the execution of the function on a small test input.

TRAVIOLI uses program instrumentation to generate a trace of *events* corresponding to reads and writes of memory-locations. In the following discussion, whenever an execution of a function reads a memory-location that the function execution has not written before, we call it an *input read-event*. An input read-event contains the address of the memory-location being read, the value being read, and the program-location where the read is performed by the function. TRAVIOLI determines the input read-events during each function execution and analyzes them to determine if the function has a traversal.

From executions of `sum` and `len` in Figures 1 and 2, respectively, one can observe that different memory-locations are read at the same program-location: `sum` reads the elements of the array `arr` at line 5 and `len` reads the `next` field of the `list` objects at line 6. This observation suggests that a traversal should satisfy the following two conditions:

C1. At least two input read-events at some program-location ℓ access different memory-locations, and

C2. the memory-locations involved in the input read-events either belong to the same object, or belong to different objects connected by a series of pointers.

Note that `addPair` in Figure 3 does not satisfy the first condition because the two elements of the array are read at different program-locations—lines 3 and 4, respectively.

The above two conditions result in a false positive for the function `third` in Figure 4. The function `third` calls `n` twice, and line 8 accesses `next` field of objects connected by a

```
1 /* Get the third element of a linked list */
2 function third(list) {
3   var node = n(list);
4   node =  n(node);
5   return node.data;
6 }
7 function n(node) {
8   return node.next;
9 }
```

Fig. 4. Another example of a non-traversing function.

```
1 /* Check if a linked list contains a value */
2 function contains(list, x) {
3   if (list === null) {
4     return false;
5   } else if (list.data === x) {
6     return true;
7   } else {
8     var tail = list.next;
9     return contains(tail, x);
10   }
11 }
```

Fig. 5. A recursive function containing a traversal.

pointer. Thus both conditions are satisfied. However, `third` does not contain a traversal, since its read-footprint is bounded to at most two linked-list nodes. The imprecision stems from the first condition, which requires two input read-events to occur at similar execution points, where two execution points are deemed similar if they have the same program-locations. This notion of similarity of two execution points is too coarse-grained. We can alleviate this problem if we say two execution points are similar if they are executing the same program-location and have identical call stacks. We capture such a state of execution in a concept called execution contexts.

Definition 1. The *execution context* of an event with respect to an execution of a function f is a sequence $(f_1 : \ell_1)(f_2 : \ell_2) \ldots (f_n : \ell_n)$, where

- f_1 is the function f,
- for each i such that $1 \leq i < n$, ℓ_i is the program-location within function f_i where f_{i+1} is invoked in the current execution, and
- the function f_n is currently executing the program-location ℓ_n to generate the input read-event.

For example, in an execution of the function `third` in Figure 4, the two input read-events at line 8 have the execution contexts `(third:3)(n:8)` and `(third:4)(n:8)` with respect to the execution of the function `third`. Unless otherwise specified, we always refer to execution contexts with respect to the execution of the function being analyzed for traversals. In order to remove the false positive for `third`, we refine the first condition for traversal as follows:

C1. At least two input read-events at some execution context access different memory-locations.

The revised condition gives no false positive for any of the previous examples. Unfortunately, this revision, which uses a fine-grained notion of similarity of execution points, introduces false negatives—it fails to detect data-structure traversals via recursive functions, such as the function `contains` defined in Figure 5.

```
1 /* Alternately add and subtract from items. */
2 function alt(obj) {
3   return p(obj.items, true, 0);
4 }
5 function p(node, flag, total) {
6   if (node != null) {
7     var value = node.data;
8     return flag ? q(node, flag, total + value)
9                 : q(node, flag, total - value);
10   } else {
11     return total;
12   }
13 }
14 function q(node, flag, total) {
15   var tail = n(node);
16   return p(tail, !flag, total);
17 }
18 function n(node) {
19   return node.next;
20 }
```

Fig. 6. Mutually recursive functions containing a traversal.

In the function `contains`, a recursive traversal occurs at line 8, but its execution does not meet condition C1 because the execution contexts of the input read-events at this program-location are different. In particular, the execution context is `(contains:8)` for the first input read-event, `(contains:9)(contains:8)` for the second input read-event, `(contains:9)(contains:9)(contains:8)` for the third input read-event, and so on. Such execution contexts become more complicated for more complex functions involving mutual recursion, such as the function `alt` in Figure 6.

The function `alt` traverses the linked list rooted at `obj.items` and alternately adds and subtracts values of its nodes to the total. The boolean `flag` passed to function `p` at line 8 decides which operation to perform, and this flag is toggled by the function `q` at line 16. Here, `p` and `q` are mutually recursive, and the traversal of the linked list occurs at line 19 after `q` calls `n` at line 15. The first time program control reaches line 19, the execution context is `(alt:3)(p:8)(q:15)(n:19)`; the second-time a different branch is taken in `p`, and thus the context is `(alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)`, and so on.

In TRAVIOLI, a key observation we make is that, despite the differences in the execution contexts of the input read-events involved in a traversal, the contexts are equivalent *modulo recursion* (i.e. after removing any cycles). Such reduced execution contexts, which we define next, are called *acyclic execution contexts* (AEC) and they are constructed as follows. For an execution context $(f_1 : \ell_1)(f_2 : \ell_2) \ldots (f_n : \ell_n)$, we first construct an *execution-context graph* consisting of a node for each unique function f_i and a special node `end`. Moreover, let `start` denote the node corresponding to f_1. For every consecutive pair $(f_i : \ell_i)(f_{i+1} : \ell_{i+1})$ in the execution context, we add a directed edge from f_i to f_{i+1} with label ℓ_i and weight i. Additionally, we add an edge from f_n to `end` with label ℓ_n and weight n. For the example in Figure 6, the execution-context graph for the second input read-event at line 19 is shown in Figure 7, where the edges are labeled by the program-locations ℓ and weights w.

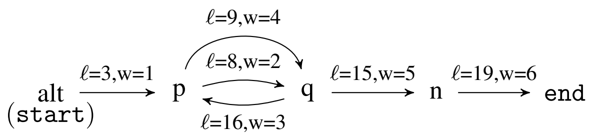

Fig. 7. Execution-context graph for the execution context `(alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)`.

Example	Execution contexts	AEC
Fig. 4, Line 8	(third:3)(n:8) (third:4)(n:8)	(third:3)(n:8) (third:4)(n:8)
Fig. 2, Line 6	(len:6) (len:6)	**(len:6)** **(len:6)**
Fig. 5, Line 8	(contains:8) (contains:9) (contains:8)	**(contains:8)** **(contains:8)**
Fig. 6, Line 19	(alt:3)(p:8)(q:15)(n:19) (alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)	**(alt:3)(p:8)(q:15)(n:19)** **(alt:3)(p:8)(q:15)(n:19)**

TABLE I
EXECUTION CONTEXTS AND AECS FOR FIRST TWO READ-EVENTS.

Definition 2. The **acyclic execution context** (AEC) of an execution context is the sequence $(f_1 : \ell_1)(f_2 : \ell_2) \dots (f_k : \ell_k)$ such that $f_1 \xrightarrow{\ell_1} f_2 \dots f_k \xrightarrow{\ell_k} f_{k+1}$ is the shortest weighted path from `start` to `end` in its execution-context graph.

For the graph in Figure 7, the acyclic execution context is `(alt:3)(p:8)(q:15)(n:19)`. As the edge weights correspond to the position of the edge in the sequence, multiple edges between two nodes are disambiguated by choosing the edge corresponding to the least recent function invocation.

Two distinct execution contexts that have the same AEC are the recursive analog of distinct iterations of a single loop. Unlike execution contexts that can grow unboundedly, AECs are bounded because the number of permutations of distinct functions in a program is finite. We found AECs to be a useful abstraction for clustering execution contexts of input read-events involved in a traversal—such an abstraction helps us to merge execution points involved in a traversal in a precise way irrespective of whether the traversal involves recursive calls or loop iterations.

Table I lists, for some example functions and program-locations (column 1), the execution contexts (column 2) and corresponding AECs (column 3) for the first two input read-events, when the functions are provided an input linked list containing at least two nodes. The first row shows that the AECs for input read-events at line 8 in the function `third` are distinct, since `third` does not contain a traversal. The last three rows show that for the functions `len`, `contains` and `alt`, multiple input read-events at the given locations have a common AEC; therefore, they are traversing functions.

We can now refine the conditions that a traversing function should satisfy in terms of AECs as follows:

C1. At least two input read-events having same the AECs access different memory-locations, and

C2. the memory-locations involved in the input read-events either belong to the same object, or belong to different objects connected by a series of pointers.

We call the AEC of such input read-events a *traversal point*. In general, a traversing function may contain more than one traversal point.

III. FORMAL DESCRIPTION

TRAVIOLI identifies the traversing functions in a program by analyzing an execution of the program. TRAVIOLI first instruments the program under analysis to generate runtime events. The instrumented program is executed with a suitable set of inputs to generate a trace of runtime events. From the generated trace, TRAVIOLI determines the input read-events for every function execution. TRAVIOLI then analyzes each sequence of input read-events to detect traversals. We next describe each of these steps formally.

A. Events and Traces

TRAVIOLI tracks reads and writes of every memory-location during an execution of a program. In a program, a memory-location can be denoted by a local variable, a global variable, a field of an object, or an element of an array. A memory-location is represented by a pair (obj, fld), where obj is the address of an object (or array), and fld is the name of a field (or index of an array element). Local variables are treated as fields of special *activation record* objects corresponding to the stack frames in which they are allocated. Global variables are treated as fields of a special *globals* object.

TRAVIOLI instruments a program to generate the following four kinds of events:

1) $\text{READ}\langle \ell, obj, fld, val \rangle$ denotes the read of a memory-location (obj, fld) at program-location ℓ. The result of the read, val, can be a scalar or the address of another object.
2) $\text{WRITE}\langle \ell, obj, fld, val \rangle$ denotes the write of a memory-location (obj, fld) at program-location ℓ. Here, val is the new value that is written to the memory-location. At function calls, write-events are generated for each argument passed to the function, where each formal parameter is treated as a local variable.
3) $\text{CALL}\langle \ell, f, a \rangle$ is an event corresponding to the invocation of function f at the program-location (i.e. call site) ℓ. Here, a is a freshly generated unique identifier for the newly created activation record object for this function invocation.
4) $\text{RET}\langle \ell, a \rangle$ is an event corresponding to a function returning to its caller. Here, ℓ is the program-location of the return instruction and a is the identifier of the current activation record, which is about to be destroyed. Note that each unique value of a appears in exactly one call and one return event in the program execution.

The execution of an instrumented program generates a trace of events. We identify the *execution of a function* started by the event $\text{CALL}\langle \ell, f, a \rangle$ by the activation record identifier a. For a function execution denoted by a, we use $\text{TRACE}(a)$ to denote the sequence of events generated by the function execution, including the call and return events that start and end the execution of the function, respectively. If a function f' is invoked during the execution of a function f with activation record a, and if this invocation creates an activation record a', then $\text{TRACE}(a')$ is a subsequence of $\text{TRACE}(a)$.

B. Read-Traces and Read-Footprints

To compute the read-footprint of a function execution, we need to determine the set of memory-locations that are read before being written during the execution. We define the *read-trace* of a function execution a, denoted by $\textsc{rtrace}(a)$, as the largest set of events e_i such that:

- $e_i = \textsc{Read}\langle *, obj, fld, * \rangle$
- $e_i \in \textsc{trace}(a)$
- $\forall j\colon (e_j = \textsc{Write}\langle *, obj, fld, * \rangle) \in \textsc{trace}(a) \Rightarrow j > i$

The third condition ensures that if there is a write to the memory-location (obj, fld) in the trace, then it must occur *after* e_i. Then, the *read-footprint* of a function execution a, denoted by $\textsc{fp}(a)$, is computed as:

$$\textsc{fp}(a) = \{(obj, fld, val) \mid \textsc{Read}\langle *, obj, fld, val \rangle \in \textsc{rtrace}(a)\}$$

C. Traversing Functions

We can now provide a formal definition of traversing functions in terms of read-footprints. Let f_X denote the execution of a function f with input X, where X represents the state of the entire program memory before such an execution, including the state of any arguments passed to f as parameters.

Definition 3. A function f is a **traversing function** if and only if the following condition holds:

$$\forall X_1 : f_{X_1} \text{ halts}, \exists X_2 : |\textsc{fp}(f_{X_2})| > |\textsc{fp}(f_{X_1})|$$

Determining if an arbitrary function is a traversing function is undecidable in general. We therefore detect potential traversals using the method described in Section II.

D. Detecting Traversals

For every function execution a and for each event e in $\textsc{trace}(a)$, we compute the execution context of e with respect to a, denoted by $\textsc{ec}(a, e)$ as follows:

- If e is the first event of $\textsc{trace}(a)$ and is of the form $\textsc{Call}\langle \ell, f, a \rangle$, then $\textsc{ec}(a, e) = \epsilon$, i.e. the empty sequence.
- If e is not the first event of $\textsc{trace}(a)$ and is generated at program-location ℓ, and if the latest call-event before e without a matching return event before e is $e' = \textsc{Call}\langle \ell', f', a' \rangle$, then $\textsc{ec}(a, e) = \textsc{ec}(a, e').(f', \ell)$, where $s.(f, \ell)$ is the sequence obtained by appending the pair (f, ℓ) to the sequence s.

This is a formal version of Definition 1 given in Section II-B.

Once we have computed the execution context of an event with respect to a function execution, we determine its acyclic execution context as per Definition 2. Let us denote the acyclic execution context of an event e with respect to a function execution a by $\textsc{aec}(a, e)$.

Next, we define a reachability relation $\overset{a}{\rightsquigarrow}$ between objects accessed in function execution a, such that $o_1 \overset{a}{\rightsquigarrow} o_n$ holds if and only if there exists a sequence $(o_1, f_1, o_2), (o_2, f_2, o_3), \ldots (o_n, f_n, val)$, such that each element of the sequence is in the read-footprint $\textsc{fp}(a)$. This relation is reflexive and transitive.

We can now formalize the conditions we check to detect if an execution a of function f contains a traversal: if there exist two input read-events $e_i = \textsc{Read}\langle \ell, obj_i, fld_i, val_i \rangle$ and $e_j = \textsc{Read}\langle \ell, obj_j, fld_j, val_j \rangle$ such that:

- $e_i, e_j \in \textsc{rtrace}(a)$
- $(obj_i, fld_i) \neq (obj_j, fld_j)$
- $\textsc{aec}(a, e_i) = \textsc{aec}(a, e_j) = \alpha$
- $obj_i \overset{a}{\rightsquigarrow} obj_j$ or $obj_j \overset{a}{\rightsquigarrow} obj_i$

then we mark the function f as a traversing function and the AEC α as a traversal point. There may be more than one acyclic execution context marked as a traversal point for a function f across one or more of the function's executions.

IV. Applications

A. Access Graphs for Visualization

TRAVIOLI can discover traversal points in functions that traverse input data structures. In order to identify the data structure being traversed, and to visualize the traversal across one or more AECs, we develop the concept of access paths and access graphs.

A memory-location in a read-footprint, which we call an *input memory-location*, can be reached from a program variable via a series of one or more fields or array indices, called an *access path.* An access path π in a function execution is a finite non-empty sequence of the form $v.k_1.k_2 \cdots k_n$, where $n \geq 0$, v is a variable name, and each k_i is either a field name or an array index. Access paths are defined recursively as follows: the access path v represents the value of the variable v before the function execution starts, and the access path $\pi.k$ represents the value stored in the field or array index k of the object whose access path is π. For example, the set of input memory-locations read by the function `third` in Figure 4 can be represented by the access paths `list`, `list.next`, `list.next.next`, and `list.next.next.data`. More than one access path may refer to the same memory-location.

Since traversing functions have read-footprints that are unbounded, we found it useful to represent the unbounded set of access paths involved in a data-structure traversal using a finite graph, called an *access graph*. Figure 8 lists access graphs for various examples used in this paper. In an access graph, nodes represent a set of values, which may be scalars or object addresses. There are two types of nodes: variable nodes and AEC nodes. A variable node with label v represents the value stored in variable v at the beginning of the function execution. An AEC node with label α represents the values read by an input read-event at AEC α. There is an edge with label k from any node n to an AEC node α if the field k of an object denoted by the n-node is read in an input read-event at the AEC α. If more than one field of objects represented by node n are read at the AEC α, then the edge from the n node to the α node is labeled with $*$. This happens when multiple elements of an array or multiple fields of an object are read at the AEC α. According to this definition, variable nodes do not have incoming edges. Moreover, all AEC nodes are reachable from at least one variable node. An AEC node is colored grey if the corresponding AEC is a traversal point.

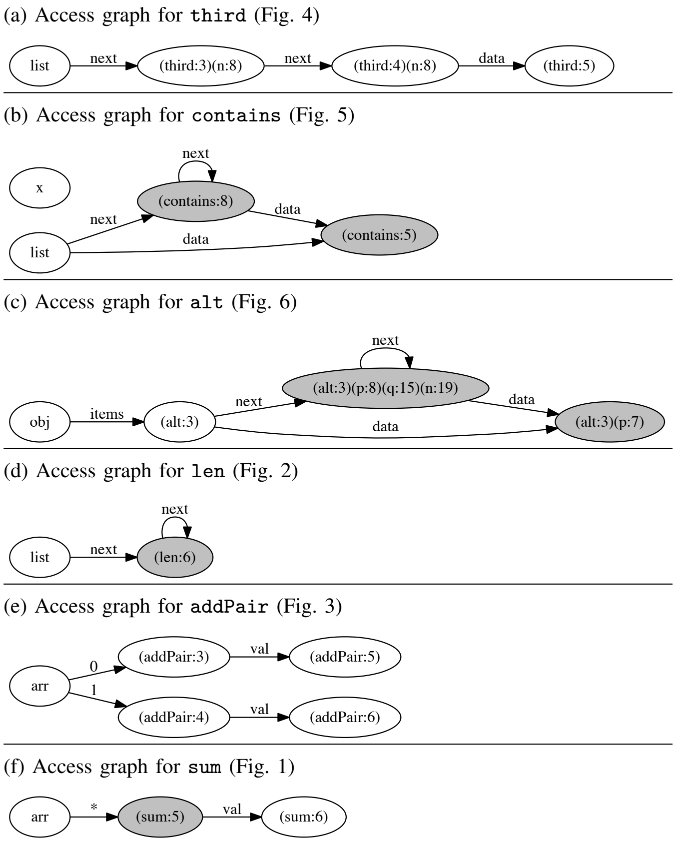

Fig. 8. Access graphs for various examples presented in the paper.

An access graph concisely captures the access paths of all input memory-locations read at each AEC. In particular, a path in the graph from a variable node v to an AEC node α corresponds to an access path that begins with v and is followed by the sequence of edge labels along the path in the access graph. For example, in Fig. 8a, the access path of an input memory-location read at AEC `(third:5)` is `list.next.next.data`. In Fig. 8b, the access graph contains a cycle. Therefore, the access paths of the input memory-locations read at AEC `(contains:5)` are `list.data`, `list.next.data`, `list.next.next.data`, and so on. In this manner, an access graph provides a bounded representation of an unbounded number of access paths.

Figures 8b, 8c and 8d represent access graphs of three functions that traverse linked lists in different ways, but the access graphs provide similar abstractions, because, in each case, the input lists are traversed via the `next` field at a single AEC. Figures 8e and 8f depict access graphs for functions that read array elements. In `addPair`, array elements are read at two distinct AECs; therefore, the graph contains two branches starting from `arr`. On the other hand, `sum` traverses the array and this is captured by the wild-card $*$ that labels the edge from `arr` to the AEC `(sum:5)`. The access paths that reach this AEC are `arr.*`, which indicate that more than one field (or in this case, more than one array index) of the variable `arr` is read at the AEC `(sum:5)`. Similarly, the access paths that are read at AEC `(sum:6)` are `arr.*.val`, which represent the `val` fields of the elements contained in the array `arr`.

We can use access graphs to determine access paths that identify the data structure being traversed. We call such an access path the *root* of the data structure. The roots are determined by identifying the shortest access path π corresponding to access-graph nodes n such that (1) there is an edge from n to a grey node and (2) there is no grey node along the path π. For example, the root of the data structure traversed in Fig. 8b is simply `list`, while in Fig. 8c the data structure being traversed is `obj.items`.

B. Performance Test Generation

Large software projects such as the Chrome browser use sophisticated frameworks to continuously perform performance regression testing [2], where the application is benchmarked at different versions in the development history; performance bugs are discovered by identifying code changes that cause statistically significant deviations in the measurements. Unfortunately, performance regression testing is not as widely used as functional testing. While there exist several code coverage tools for measuring completeness of functional tests, there is a lack of tool support to identify code modules that should be the focus of performance tests.

Previous research suggests that hard-to-detect performance bugs are often exposed when applications are executed with large-scale inputs and/or with special input values [17]. Our technique can be used to identify the former case. If functional unit tests for an application are available, we can use TRAVIOLI to find functions that traverse input data structures and assist developers in constructing performance unit tests that force long traversals of the input data structures.

We next illustrate the process of creating a performance unit test from a functional unit test using TRAVIOLI. Consider the function `alt` in Fig. 6 and a unit test in Fig. 9. The unit test, `altTest`, first invokes the function `makeRange` at lines 21 and 22 to create sample objects containing lists with the first few natural numbers. Lines 23 and 24 contain calls to `alt` and assertions to ensure that the result matches the expected total. If this unit test is provided to TRAVIOLI, the following report is generated:

```
Data structure 'obj.items' in function 'alt':
- Traversal point: (alt:3)(p:8)(q:15)(n:19) [max 6x]
  - Absolute AECs for traversal events:
    1. (altTest:23)(alt:3)(p:8)(q:15)(n:19)
    2. (altTest:24)(alt:3)(p:8)(q:15)(n:19)
  - Values last written at these absolute AECs:
    1. (altTest:21)(makeRange:14)
    2. (altTest:22)(makeRange:14)
- Traversal point: (alt:3)(p:7) [max 6x]
  ... <trimmed> ...
```

TRAVIOLI detects two traversal points for the data structure `obj.items` in the function `alt`, corresponding to the program-locations that access `node.next` and `node.data`. Note that the traversals are identified when analyzing the executions of the function `alt` and not `altTest`, as the linked list is not an external input to the latter; therefore, the AECs identifying the traversal point are with respect to the execution of `alt`. TRAVIOLI collects and reports three types of information for each traversal point α in function f. First, a traversal-point

```
/* Creates a linked list with numbers 1..N */
function makeRange(N) {
  var node, rangeList = null;
  for (var i = N-1; i >= 0; i--) {
    node = {data: i, next: rangeList};
    rangeList = node;
  }
  return rangeList;
}
/* Test the alt() function */
function altTest() {
  var o1 = {items: makeRange(3)};
  var o2 = {items: makeRange(6)};
  assert(alt(o1) ===  2);
  assert(alt(o2) === -3);
}
```

Fig. 9. Unit test for the `alt` function from Fig. 6.

```
/* Execute 'alt' over a large linked list. */
var benchmark = new Benchmark.Suite;
var bigObj = {items: makeRange(100000)};
benchmark.add('alt-perf', function () {
  return alt(bigObj);
}).run();
```

Fig. 10. Performance test for the `alt` function from Fig. 6.

counter keeps track of the maximum number of times f has executed α. In our example, the longest traversal corresponds to the second invocation of `alt`, where the traversal point `(alt:3)(p:8)(q:15)(n:19)` is observed six times; therefore, the report annotates this AEC with `[max 6x]`. Second, every event e is associated with an absolute AEC, which is equivalent to an AEC with respect to a top-level entry function (such as `main` in some languages) or a configurable harness function from the test framework. These AECs are listed under the header `"Absolute AECs for traversal events"` in the report; the example shown above lists AECs with respect to the harness `altTest`. This information aids in determining how traversing functions are invoked by client code or test cases. Third, for all input memory-locations m read at an event e, we associate an event e' at which m was last written-to before e. When reporting traversals, TRAVIOLI lists the absolute AECs of such write events for memory-locations involved in traversals under the header `"Values last written at"`. In example shown above, the memory-locations read at the first traversal point are the `next` fields of the linked-list nodes; therefore, the report lists the two distinct absolute AECs by which these fields were populated before the traversal. This information aids in identifying the point of construction or last modification of elements of a data structure that is subsequently traversed. In general, the relationship between reads and writes in a traversal is many to many: data-structure elements that were last modified at distinct locations may be traversed at the same absolute AEC, while a data-structure whose elements are populated at a single absolute AEC may be subsequently traversed at multiple AECs.

In our experiments, we found that this report provided useful information to track where and how data structures were constructed and provided as inputs to traversing functions. If the goal of a developer is to write a performance unit test exercising a traversal point, they can use this information to write a test that will increase the value of the traversal-

```
/* Does 'list' contain everything in 'arr'? */
function containsAll(list, arr) {
  for (var i = 0; i < arr.length; i++) {
    var item = arr[i];
    if (contains(list, item) == false) {
      return false;
    }
  }
  return true;
}
```

Fig. 11. A function that redundantly traverses a list.

point counter by several orders of magnitude. Fig. 10 shows a sample performance test for the running example, using the Benchmark.js API [1]. To verify that this test does indeed have a large read-footprint, we can run it through TRAVIOLI to generate a report that will annotate the AEC `(alt:3)(p:8)(q:15)(n:19)` with `[max 100000x]`.

C. Detecting Redundant-Traversal Bugs

TRAVIOLI can also be used to detect *redundant traversals*, such as the traversal in the function `containsAll` shown in Figure 11. The function `containsAll` takes as input a linked list `list` and an array `arr` and returns `true` if an only if all items in the array are also present in the list, by repeatedly invoking the `contains` function defined in Figure 5. The list is traversed multiple times without any change to its data—this is a case of redundant traversal. If the list contains n elements and the array is of length m, then the worst-case complexity of `containsAll` is $\mathcal{O}(mn)$. Such instances are often indicative of performance bugs and can be fixed by using different data structures (such as hashed sets) or caching. TRAVIOLI found two such instances in popular JavaScript projects, which were acknowledged by their developers as performance issues.

In order to determine if a traversal in a function is redundant, we need to analyze the sequence of concrete memory-locations (i.e. actual memory addresses) read at a traversal point of the function. If the sequence contains repeated contiguous subsequences, then we know that the memory-locations in these contiguous subsequences are traversed repeatedly. We then say the function has a redundant traversal. Formally, if the sequence of memory-locations read at a traversal point can be partitioned into the contiguous subsequences $\beta_1, \beta_2, \ldots, \beta_k$ where $k \geq 2$ and for each $1 \leq i, j \leq k$, either β_i is a prefix of β_j or β_j is a prefix of β_i, then the sequence of memory-locations indicate a possibly redundant traversal.

For example, if a, b and c are concrete memory-locations, then the sequence of reads $abcaba$ can be partitioned into repeating contiguous subsequences $(abc)(ab)(a)$ indicating redundant traversals. On the other hand, the sequence $abcacab$ is partitioned as $(abc)(ac)(ab)$ and does not indicate a redundant traversal because ac is not a prefix of ab and vice versa.

Consider the execution of `containsAll` on an input linked list `list` containing the elements `['a', 'b', 'c']` and an array `arr` containing `['c', 'b', 'a']`. The report generated by TRAVIOLI when analyzing this execution indicates the presence of a potentially redundant traversal and includes the lengths of the repeating subsequences observed at the AEC corresponding to the redundant traversal, as follows:

```
Data structure 'list' in function 'containsAll':
- Traversal point: (containsAll:5)(contains:8) [6x]
  - Redundant subsequences: [3, 1, 2]
  - Absolute AECs for traversal events:
  ... <trimmed> ...
```

In general, a redundant traversal can be detected by a memory-location sequence as short as *aba* or *aab*; therefore, TRAVIOLI can detect redundant traversals from functional unit tests alone. Moreover, TRAVIOLI can detect redundant traversals in functions that use recursion, such as the example in Figure 11, which could not be detected using previous approaches [19, 20].

V. EVALUATION

We have implemented TRAVIOLI using the Jalangi framework [25] for instrumenting JavaScript programs. We evaluate TRAVIOLI on a set of five open-source JavaScript projects. The projects were chosen because they are widely used, they have comprehensive unit tests that can be launched from command-line using Node.js [9], and they represent a variety of scenarios where data-structure performance may be important. The projects include `d3-collection` [4], a data-structure library used in the popular D3 [3] visualization toolkit, `immutable-js` [7], an immutable data-structure library developed by Facebook, `d3-hierarchy` [5], which provides algorithms for visualizing hierarchical data-sets, `express` [6], a server-side web framework, and `mathjs` [8], an extensive math library. We analyze the `matrix` module of `mathjs`. The source code of TRAVIOLI has been made publicly available at https://github.com/rohanpadhye/travioli, along with the scripts to reproduce the experiments described in this section.

Table II provides an overview of experiments performed on a MacBook Pro with an Intel Core i7-4770HQ processor and 16GB RAM running OS X 10.10 and Node.js v4.4.0. All listed run-times are in seconds. Column 1 lists the candidate projects, column 2 lists the number of unit tests in their test suites, column 3 reports the running time of the corresponding test suites, and column 4 reports the running time of the instrumented test suites, including the time to instrument the source files (project + dependencies) and the time to generate events. Column 5 lists the number of events that are generated and subsequently analyzed. Columns 6–8 report the time required to analyze these events, the number of function executions analyzed for traversals, and the number of unique functions for which access graphs are generated. Although we compute the read-trace for all function executions, we exclude analysis of functions from the project's dependencies or test suites. Columns 9–11 report the results of traversal detection: the number of traversing functions, the number of distinct access paths identified as roots of data structures (cf. Section IV-A), and the number of distinct AECs marked as traversal points. Columns 12–14 repeat this information for redundant traversals (cf. Section IV-C).

For each candidate project, the instrumented tests as well as the analysis of traces completed within few minutes. We evaluate the quality of the traversals reported by answering four research questions:

RQ1. *Do the traversals reported by* TRAVIOLI *contain false positives?*

RQ2. *Can we generate performance tests for the traversals reported by* TRAVIOLI*?*

RQ3. *Do the redundant traversals reported by* TRAVIOLI *contain false positives?*

RQ4. *Do the redundant traversals reported by* TRAVIOLI *correspond to performance issues?*

Methodology: We answer RQ1 and RQ2 by manually evaluating a subset of the traversals reported by TRAVIOLI. For each candidate project, we randomly sample up to 10 access paths reported as roots of data structures being traversed, and randomly pick one reported traversal point for each access path. If a reported traversal point does not correspond to a traversing function within the library, we classify it as a *false positive*. In all other cases, the traversal point lies within a traversing function as per Definition 3, and is thus a true positive. We attempt to generate performance tests for these functions such that the counter of the AEC corresponding to the traversal point increases by a factor of 100 (see Section IV-B). However, this is sometimes not possible. A traversing function may be private to the library to which it belongs, and this library may use domain-specific constraints to ensure that the function receives inputs of only a bounded size. We classify such cases as *restricted* traversals. In such instances, we cannot write a performance test using only the external public API, and indeed this is acceptable since single restricted traversals cannot become a performance bottleneck.

Similarly, we answer RQ3 and RQ4 by manually evaluating a random subset of the traversal points that are reported as *redundant*. If the reported traversal point was not really redundant, we mark it as a *false positive*. If the traversal was redundant but the input was bounded in size, we mark it as a *restricted* traversal. We classify the remaining cases as either *bugs*—when the implementation performs more work than an optimal algorithm—or *necessary redundancies*—when the optimal algorithm necessarily requires repeated traversals of a data structure (e.g. matrix multiplication).

RQ1: Of the 50 traversal points that were randomly sampled across all candidate projects, we found only two false positives: one in `immutable-js` and another in `express`. In `immutable-js`, an array data structure was incorrectly reported to be traversed. The false positive resulted from a related traversal of a hash-map that mapped strings to integer values; the resulting integers were used as indices to access a single element of different arrays. The array accesses occurred within the same loop that traversed the hash-map, and in at least two iterations a common array was accessed at the same AEC; therefore, the conditions that TRAVIOLI checks for detecting traversals were satisfied. In `express`, one traversal point was in the test suite itself, in a function that was supplied as a callback parameter to `express`. Since the traversal was not really part of `express` we marked this as a false positive.

RQ2: Of the 48 true traversals in our sample, we found three instances of *restricted* traversals; all three belonged to `immutable-js`. The corresponding traversing functions

Application	Test Suite		Instrumented Tests		Analysis			All Traversals			Redundant Traversals		
	Test Cases	Run. Time	Run. Time	Events Logged	Run. Time	Function Invocations	Unique Func.	Unique Func.	Unique Roots	Unique AECs	Unique Func.	Unique Roots	Unique AECs
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
d3-collection	233	0.21	7.95	593,752	3.88	1,340	37	13	15	36	0	0	0
immutable-js	418	0.65	81.12	11,677,694	149.69	260,642	513	239	460	2,859	45	62	106
d3-hierarchy	49	0.18	10.14	1,021,083	6.40	5,523	50	20	23	88	1	1	1
mathjs:matrix	357	0.59	43.80	4,341,074	44.33	26,931	282	128	226	2,261	31	13	1,444
express	696	2.12	81.09	7,454,087	91.52	53,382	158	50	81	1,847	2	2	2

TABLE II
OVERVIEW OF EXPERIMENTS CONDUCTED TO EVALUATE TRAVIOLI. ALL TIMES ARE IN SECONDS.

Application	False Positives	Restricted Traversals	Perf. Tests Generated	Total
d3-collection	0	0	10	**10**
immutable-js	1	3	6	**10**
d3-hierarchy	0	0	10	**10**
mathjs:matrix	0	0	10	**10**
express	1	0	9	**10**
Total	**2**	**3**	**45**	**50**

TABLE III
EVALUATION OF SAMPLED TRAVERSAL POINTS.

Application	False Positives	Restricted Traversals	Necessary Redun-dancies	Perf. Bugs	Total
d3-collection	0	0	0	0	**0**
immutable-js	4	6	0	0	**10**
d3-hierarchy	0	0	0	1	**1**
mathjs:matrix	0	3	7	0	**10**
express	0	1	0	1	**2**
Total	**4**	**10**	**7**	**2**	**23**

TABLE IV
EVALUATION OF SAMPLED REDUNDANT TRAVERSAL POINTS.

perform traversals of input arrays. However, these functions are only used to traverse arrays contained in nodes of a bit-partitioned vector trie. Each node in such a trie can have up to a maximum of 32 child nodes. These child nodes are stored in an array, whose traversal was reported by TRAVIOLI. The array traversal counter can never exceed 32.

We summarize the results of the evaluation of RQ1 and RQ2 in Table III. The false positive rate in our evaluation was 4%. For the true positives, we could construct performance tests in 93.75% of the cases.

RQ3: From the reported redundant traversals, we sampled 10 data structures and one corresponding traversal point from both `immutablejs` and `mathjs`. `d3-hierarchy` and `express` contained fewer than 10 reports of redundant traversals and we analyzed all of those cases. No redundant traversal was reported for `d3-collection`. We manually analyzed a total of 23 redundant traversals. All false positives were in `immutable-js`. The sequence of memory-locations read at the reported traversal points did contain repeated contiguous subsequences, but this was specific to the particular inputs in the test suites. The corresponding traversing functions do not perform redundant computations in general.

RQ4: Of the 19 sampled redundant traversals that were true positives, we found 10 to be *restricted* traversals. For example, the implementation of maps in `immutable-js` uses `ArrayMap` with linear-time lookup only when the number of elements is less than 8; for larger maps the implementation switches to using hash-tables with constant-time lookup. In `express`, one reported redundant traversal was restricted because the traversing function can only ever be invoked internally with a list of HTTP methods (e.g. `GET`, `POST`), of which only 26 are supported; therefore, this function does not lead to performance issues. In `mathjs`, all reported redundant traversals belonged to algorithms that required repeated traversals, such as matrix multiplication, and thus were not classified as bugs.

Two of the reported redundant traversals were real performance bugs—they were confirmed by the developers. In `d3-hierarchy`, TRAVIOLI found a bug in the implementation of binary tree-maps, which are a visualization of hierarchical data as rectangles that are repeatedly partitioned into two sets. The implementation partitions an array of numbers by computing an index such that the sums of the left and right sub-arrays are approximately equal. This process is recursively repeated for each partition, resulting in a binary tree. We detected, from a simple unit test, that the algorithm to find the index to partition the array performed redundant traversals at each step to compute the sums of the sub-arrays. We were able to show that in the worst-case the implementation had complexity $\mathcal{O}(n^2)$. We reported and fixed this bug (see https://github.com/d3/d3-hierarchy/issues/44), by computing the sums of all prefixes of the input array ahead-of-time, and using a binary search to find the partition index at each step. The fixed implementation is $\mathcal{O}(n \log n)$ in the worst-case, and provides about a $20\times$ speed-up for a binary tree-map with 1,000 nodes.

The second bug was found in `express`. When an `express` application is configured to support m URL patterns with n handlers using a particular API, the list of URL patterns is redundantly traversed once per handler to construct a regular expression that combines all patterns. As regex compilation is expensive, this implementation may lead to longer start-up times for some applications. We reported this as a performance issue, which was subsequently acknowledged by the developers (see https://github.com/expressjs/express/issues/3065).

Table IV summarizes the evaluation of redundant traversals. 17.4% of the reports were false positives. 52.6% of the true positives were restricted, and 36.8% were benign. TRAVIOLI found two real performance bugs that have been confirmed by the developers.

VI. DISCUSSION

Human effort: The performance tests for the traversals sampled in our experiments were manually constructed by one of the authors, who required less than two hours per project (i.e. up to 10 tests), despite having no prior experience with the projects' internal source code or external API.

Completeness: TRAVIOLI uses dynamic analysis; therefore, it can only detect and report traversals if they occur during program execution. We cannot precisely evaluate TRAVIOLI for *false negatives*, because it is not possible to statically determine all traversal points, and it is not feasible to manually evaluate all candidate acyclic execution contexts. Our technique also does not consider *write-only* traversals, such as program functions that construct data structures from primitive values (e.g. parsing source code into an abstract syntax tree).

Threats to validity: Due to the manual effort required, we only evaluated a randomly sampled subset of the reported traversals; the results are therefore subject to sampling error. Further, our experiments indicate that the distribution of false positives, restricted traversals, and necessary redundancies is not uniform; therefore, our results may not generalize to project domains outside of those we considered for evaluation. However, we are encouraged by the fact that we could easily construct performance tests for arbitrary projects as well as discover asymptotic performance bugs in two of them.

VII. RELATED WORK

1) Redundant computation bugs: Clarity [20] uses static analysis to detect program functions in which an $\mathcal{O}(n)$ traversal occurs $\mathcal{O}(m)$ times redundantly. As our analysis is dynamic, we can determine if repeated traversals are redundant at a finer granularity. For example, if a binary-search-tree is repeatedly queried for different values, we do not report a redundancy if at least two traversals follow different paths in the tree. Clarity conservatively assumes all conditional branches to be equally likely, and thus cannot make such fine-grained distinctions automatically. Clarity therefore uses source-level annotations to recognize operations on standard Java collections that have sub-linear average-time complexity. However, Clarity's static analysis is a sound over-approximation, while our dynamic analysis is subject to false negatives.

Toddler [19] uses dynamic analysis to detect similar memory access patterns at the same execution context. It detects redundancies by analyzing the execution of long-running performance tests and extracting similarities in memory accesses across loop iterations. Our use of AECs and object connectivity allow us to detect traversals from as little as two iterations, and therefore we can detect redundant traversals using unit tests alone. Moreover, acyclic execution contexts enable the detection of *recursive* data-structure traversals, which is not supported by either of these tools.

MemoizeIt [14] uses dynamic analysis to detect functions whose computation can be memoized—this includes a special case of redundant traversals where the repeating subsequences are exactly equal. MemoizeIt can therefore detect the type of bug we found in `express`, but not the bug we found in `d3-hierarchy`.

2) Performance test generation: SpeedGun [22] generates performance regression tests for multi-threaded programs to identify code changes that influence the amount of synchronization required. PerfPlotter [12] uses symbolic execution to generate distributions of a program's performance under different inputs. WISE [11] uses symbolic execution to automatically generate tests that exercise worst-case behavior. These techniques aim to automatically generate test programs or inputs that exercise special performance characteristics. Our goal is not to automate test generation, but to identify program functions that traverse data structures and to aid developers in writing performance tests that exercise these traversals.

3) Data-structure analysis: A number of techniques have been developed to analyze data structures using dynamic analysis. HeapViz [10] summarizes relationships between Java collections to provide a concise visualization of the heap. MG++ [28] generates representations of dynamically evolving data structures. Pheng and Verbrugge [21] measure the number of data structures created and modified over time in Java programs. Laika [13] detects data structures in executing binaries using Bayesian unsupervised learning. DSI [31] identifies pointer-based data structures in C programs. Raman and August [23] detect recursive data structures and profile structural modifications in order to measure their stability.

Similarly, several static analysis techniques aim to discover abstract representations of data structures used in a program, and this body of work usually falls into the category of *shape analysis* [16]. Sophisticated frameworks can be used to prove complex data-structure invariants [24].

In all these techniques, the central theme has been identifying the type of data structures or their representation in program memory, and not on identifying functions that *traverse* these data structures to perform work.

4) Execution Contexts and AECs: In dynamic analysis, execution indexing [32] allows uniquely identifying a point in a program execution. Such execution indices are too fine-grained for TRAVIOLI. The problem of reasoning about an unbounded number of calling contexts in recursive programs is well-known in the field of static analysis [26, 27]. Our approach of constructing AECs by removing cycles in execution contexts is similar to the approach employed by Whaley and Lam [30] for context-sensitive pointer analysis, where connected components in the call graph are collapsed to a single node. A subtle difference is that we retain the sequence of functions on paths from the entry of a connected component to its exit; therefore, the resulting AEC is a valid sequence of call sites that can be used for stack-trace debugging.

5) Access Graphs: Access graphs were first used in a static liveness analysis [18] to represent an unbounded set of heap-memory locations that may be live at a program point. Our access graphs are similar in that a node can represent a regular pattern of access paths. However, we distinguish nodes based on AECs rather than program-locations as in the original formulation; therefore, our access graphs are context-sensitive.

ACKNOWLEDGMENT

We would like to thank Caroline Lemieux for her valuable comments on the paper. This research is supported in part by an Okawa Foundation Research Grant, and NSF grants CCF-1409872 and CCF-1423645.

REFERENCES

[1] Benchmark.js. https://benchmarkjs.com. Retrieved: August 2016.

[2] Chrome performance dashboard. https://chromeperf.appspot.com. Retrieved: August 2016.

[3] D3: a JavaScript library for visualizing data with HTML, SVG, and CSS. https://d3js.org. Retrieved: August 2016.

[4] d3-collection: Handy data structures for elements keyed by string. https://github.com/d3/d3-collection. Retrieved: August 2016.

[5] d3-hierarchy: 2d layout algorithms for visualizing hierarchical data. https://github.com/d3/d3-hierarchy. Retrieved: August 2016.

[6] express.js: Fast, unopinionated, minimalist web framework for node. https://github.com/expressjs/express. Retrieved: August 2016.

[7] immutable.js: Immutable persistent data collections for Javascript. https://github.com/facebook/immutable-js. Retrieved: August 2016.

[8] Math.js: An extensive math library for JavaScript and Node.js. https://github.com/josdejong/mathjs. Retrieved: August 2016.

[9] Node.js. https://nodejs.org. Retrieved: August 2016.

[10] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[11] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 49–60, New York, NY, USA, 2016. ACM.

[13] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[14] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015. ACM.

[15] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphvizopen source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[16] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[17] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[18] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[19] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[20] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 369–378, New York, NY, USA, 2015. ACM.

[21] Sokhom Pheng and Clark Verbrugge. Dynamic data structure analysis for Java programs. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 13–25, New York, NY, USA, 2014. ACM.

[23] Easwaran Raman and David I. August. Recursive data structure profiling. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 5–14, New York, NY, USA, 2005. ACM.

[24] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[25] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[26] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Muchnick and Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc, 1981.

[27] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[28] V. Singh, R. Gupta, and I. Neamtiu. MG++: Memory graphs for analyzing dynamic data structures. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 291–300, March 2015.

[29] Steven S. Skiena. *The Algorithm Design Manual*. Springer London, second edition, 2009.

[30] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[31] David H. White, Thomas Rupprecht, and Gerald Lüttgen. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 259–269, New York, NY, USA, 2016. ACM.

[32] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.