MemInsight: Platform-Independent Memory Profiling for JavaScript

Simon Jensen, Snowflake Computing

Manu Sridharan, Samsung Research America

Koushik Sen, University of California, Berkeley

Satish Chandra, Samsung Research America

Technical Report SRA-FCS-2015-001
March 2015

SAMSUNG

SAMSUNG RESEARCH AMERICA

MemInsight: Platform-Independent Memory Debugging for JavaScript

Simon Holm Jensen*
Snowflake Computing, USA
simon@hjensen.net

Koushik Sen*
UC Berkeley, USA
ksen@cs.berkeley.edu

Manu Sridharan
Samsung Research America, USA
m.sridharan@samsung.com

Satish Chandra
Samsung Research America, USA
schandra@acm.org

ABSTRACT

JavaScript programs often suffer from memory issues that can either hurt performance or eventually cause memory exhaustion. While existing snapshot-based profiling tools can be helpful, the information provided is limited to the coarse granularity at which snapshots can be taken. We present MEMINSIGHT, a tool that provides detailed, time-varying analysis of the memory behavior of JavaScript applications, including web applications. MEMINSIGHT is platform independent and runs on unmodified JavaScript engines. It employs tuned source-code instrumentation to generate a trace of memory allocations and accesses, and it leverages modern browser features to track precise information for DOM (document object model) objects. It also computes exact object lifetimes without any garbage collector assistance, and exposes this information in an easily-consumable manner for further analysis.

We describe several client analyses built into MEMINSIGHT, including detection of possible memory leaks and opportunities for stack allocation and object inlining. An experimental evaluation showed that with no modifications to the runtime, MEMINSIGHT was able to expose memory issues in several real-world applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Memory profiling, leak detection

1. INTRODUCTION

JavaScript has become a popular, mainstream programming language. It is the *lingua franca* of the web and is mak-

---

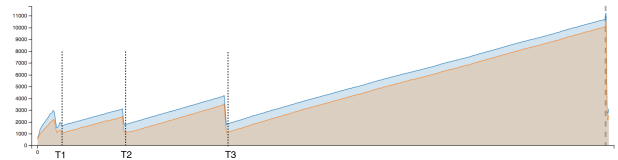*This work was performed while at Samsung Research America.

Figure 1: Number of objects in memory for three moves in *annex*, as shown by MemInsight. The x-axis represents logical time.

ing inroads into other settings such as servers via `node.js`. However, developer tools for JavaScript still lag behind those for better-established languages like C and Java, including tools for memory profiling.

Though free of traditional memory leaks, JavaScript programs often suffer from memory issues seen in other garbage-collected languages. It is common to find avoidable instances of *drag*, i.e., objects reachable in the heap past their last use [31]; if such objects never become unreachable, they are considered *leaked*. JavaScript programs also suffer from *churn*, i.e., frequent heap allocation of short-lived objects and *bloat*, i.e., wasteful representation of data. It is common to find mistakes in the way JavaScript objects and the objects in the browser's document object model (DOM) interact, causing insidious DOM leaks. Moreover, although JavaScript frameworks such as jQuery [15] are intended to make web programming simpler, it is easy to introduce memory problems via incorrect use of framework APIs.

As a concrete example of drag, consider Figure 1, which shows the number of objects in memory over time in *annex* [2], a Reversi-style game implemented as a web application. T1, T2 and T3 denote the start of the machine's computation of its move in response to the human player's first three moves, respectively. The blue (upper) curve shows the number of objects reachable in the heap; the orange (lower) curve shows the number of *stale* objects reachable in the heap. An object is stale if it is reachable in the heap but will not be used again in the current execution. Such objects can waste space, since they cannot be reclaimed by the garbage collector. Although the memory for computation of the machine's move is de-allocated right after the computation, the peak memory consumption per move is increasing, and much of the memory is stale. If there were a way to release this stale memory early, or not allocate it to begin with, then the program's peak memory usage could be much lower. (In

fact, using our tool, we were able to refactor this program to reduce its peak memory usage substantially; see Section 5.)

A common approach to debugging memory issues in JavaScript applications is to take periodic snapshots of the heap, as implemented, for example, in the Chrome Developer Tools [5] (CDT). CDT allows for visualizing various useful properties of a heap snapshot, e.g., a count of objects of each type (as determined by its constructor) and the access paths to those objects. The recommended way to diagnose memory issues with CDT is to compare (diff) a series of snapshots and see if the counts of certain types of objects unexpectedly increase, indicating a possible memory leak.

While useful, the snapshot approach suffers from the key limitation that it misses many transient behaviors that occur *between* snapshots. For example, if snapshots were manually taken between computer moves for the execution shown in Figure 1 (e.g., at times T2 and T3), they would completely miss the staleness problem in this app. In fact, with snapshots alone, one cannot determine whether an object is stale or not. Snapshots may also miss object churn issues, as a large number of short lived objects may get allocated and collected between successive snapshots.

For C and Java developers, a number of techniques and tools have been developed (e.g., Valgrind [25]) that go beyond heap snapshots to provide very detailed visibility into a program's memory usage and possible problems [3, 16, 21, 23, 27, 37, 39–41]. These techniques often modify language runtimes to monitor memory behaviors and apply various heuristics such as heap growth and object drag to detect memory problems. However, there are several challenges in simply porting these techniques and tools to JavaScript and web applications. First, JavaScript programs can be written for various platforms such as desktop browsers, browsers on mobile operating systems such as Tizen (www.tizen.org) the `WebViewer` component of iOS, and `node.js` all running different JavaScript engines having different native APIs. Therefore, it is not possible to develop a universal JavaScript memory debugging tool, which works for all platforms, by modifying a single JavaScript engine. Second, JavaScript engines of browsers can access, create, and manipulate DOM components of a browser. Such interactions allow objects to move from the JavaScript world to the DOM world and vice versa. Such data interactions across the language boundary create unique challenges when one cannot monitor memory behaviors by modifying a JavaScript engine.

To address these challenges, we have developed MEM-INSIGHT, a browser-independent extensible memory debugging tool for JavaScript and web applications. The tool works by instrumenting JavaScript programs and can run on any platform that provides a standard JavaScript engine. The overall architecture of MEMINSIGHT is shown in Figure 2. First, JavaScript code (A) is instrumented, such that exercising instrumented code (B) yields a trace (C) of operations relevant to memory analysis, described in Section 3.1. To trace DOM-related memory behaviors, MEMINSIGHT uses modern browser features to asynchronously observe DOM mutations, and employs select additional modeling to record informative allocation sites for nearly all DOM nodes in practice (details in Section 3.4). A separate *lifetime analysis* recovers object lifetime information from the trace by simulating the application's heap and combining reference counting with the Merlin algorithm [14] (see Section 3.2). Both the lifetime analysis and trace generation phases are

carefully designed to handle unusual cases arising due to uninstrumented code and native library code (Section 3.3). The lifetime analysis produces an enhanced trace (D) that includes object unreachability events. Finally, a number of client analyses compute useful additional information based on the enhanced trace, such as potential memory leaks, drags, churns, and opportunities for stack allocation and object inlining (Section 4). The results are shown in a web-based GUI. MEMINSIGHT is fully extensible: the enhanced trace generated by MEMINSIGHT can be queried to implement other client analyses.

We observed that simply reporting a memory problem is often not sufficient to effectively debug the problem. MEM-INSIGHT provides useful diagnostic information, call tree and access paths, for the objects responsible for a memory problem. A call tree shows us the calling context in which the problematic objects got allocated and the access paths reveal the chain of objects that is holding the problematic object and is preventing it from getting garbage collected. These information help developers to effectively understand and fix the memory problems reported by MEMINSIGHT.

A key advantage of MEMINSIGHT is that it is browser agnostic and requires no modification to a JavaScript runtime. As such the tool is usable across a variety of JavaScript environments, including desktop browsers, browsers on mobile operating systems like Tizen, and `node.js` on the server. Beyond the need for portability, building the tool without runtime modifications brings advantages in maintainability and deployability. JavaScript runtimes and browsers evolve very quickly, making maintaining a modified runtime while retaining support for the latest browser features an enormous challenge. Deployment also becomes difficult with a modified runtime, as users must download and use a custom browser build for profiling purposes. For all these reasons, we focused on developing a browser-independent, instrumentation-based approach to memory profiling.

In MEMINSIGHT, we addressed several challenges that arose due to our pragmatic decision to keep MEMINSIGHT browser independent:

- Reasoning about an object's staleness requires determining when the object becomes unreachable. However, JavaScript does not provide any built-in method for determining when an object is garbage collected (from which unreachability times can be computed [14]). Hence, we had to reconstruct object lifetime information in MEMINSIGHT with no support from the JavaScript runtime.

- Tools such as Valgrind typically assume that they are able to observe *all* memory accesses in a program. For web applications, this assumption is very hard to satisfy without browser modification, due to various native libraries and the DOM. Furthermore, a user might wish to purposely exclude certain frameworks from instrumentation. We designed MEMINSIGHT so that it is resilient to passing of objects back and forth between instrumented and uninstrumented code.

- The DOM and its manipulation via JavaScript present special challenges. DOM manipulation can be performed via a wide variety of APIs that would be difficult to model exhaustively. Also, the notion of staleness for a DOM node should take into account whether it is visible on the web page, which is not evident when just tracing memory accesses.
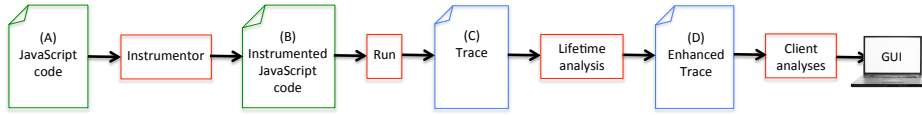
Figure 2: MemInsight tool chain

• Finally, JavaScript is a complex language, and dealing with features such as closures, constructors, getters and setters, etc. is not straightforward, particularly when instrumenting source code rather than a normalized intermediate representation within a runtime.

To the best of our knowledge, no previous work has shown how to address all of these challenges together in a realistic and practical tool that runs on unmodified browser runtimes. We consider this as *our key contribution.*

We performed an experimental evaluation to test both the overhead and usefulness of MemInsight on a variety of benchmarks. MemInsight incurred a reasonable 41.7X mean slowdown for the compute-intensive Octane benchmarks, with the slowdown being hardly noticeable for more typical web applications. Thus far, MemInsight has been used to discover or confirm memory issues in ten web and `node.js` applications, with several issues discovered by a product group at Samsung using the tool. In three cases, we submitted patches to fix discovered issues that were merged by the developers.

MemInsight is publicly available at `https://github.com/Samsung/meminsight`. A replication package for MemInsight has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations. We believe that MemInsight can be modified and extended easily even if a researcher/developer is not an expert in the internals of a JavaScript engine.

The rest of the paper is organized as follows. Section 2 gives a detailed overview of debugging a memory issue with MemInsight. The different phases of MemInsight are outlined in Sections 3 and 4, as described above. Sections 5 and 6 respectively present case studies showing MemInsight's effectiveness and a quantitative evaluation. Finally, Section 7 discusses related work, Section 8 discusses limitations, and Section 9 concludes.

2. DEBUGGING WITH MEMINSIGHT

In this section, we highlight the use of MemInsight from a user's perspective. We present an example illustrating the types of memory leaks often present in web applications, and show how MemInsight's features combine to help diagnose the problem.

A small jQuery-based example is shown in Figure 3, based on a real drag issue found in our shopping list case study (see Section 5). In function `f()`, a new *div* DOM node is allocated (into variable `newDiv`) (line 2), made to contain the text `"Hello world"` (line 3), and attached to an existing DOM node identified by the ID `'contents'` (line 7). An event handler function is attached to the newly allocated node (lines 4–6), which changes the background color when a user clicks on the *div*. In function `g()`, the developer removes this new *div* from the DOM by assigning an empty string to the `innerHTML` property of the `'contents'` node (line 11).

```
1   function f() {
2     var newDiv = $('<div/>');
3     newDiv.html("Hello world");
4     newDiv.click(function () {
5       newDiv.css("backgroundColor", "red");
6     });
7     newDiv.appendTo('#contents');
8   }
9
10  function g() {
11    document.getElementById('contents').innerHTML = '';
12  }
```

Figure 3: `scr_orig.js`, a small leak example

Alas, there is a memory problem. The jQuery framework caches event handlers associated with DOM nodes, and clears the cache only when a node is removed using a proper jQuery API call, e.g. (in this case) `$('#contents').empty()`. Since the developer removed the node via `innerHTML` instead, the event handler closure remains in jQuery's cache and becomes *stale*, as it does not get used again.

Since the event handler uses variable `newDiv` (line 5), a reference to the *div* element remains in its closure, causing the DOM element itself to also drag; this is known in the developer community as *leaking* of DOM nodes, as it a common issue in web applications. (We will use the terms leak and drag interchangeably.) Closure-related drags are common in JavaScript programs, as even local variables unrelated to the function for which the closure is created can get retained by the closure, extending their lifetimes inadvertently.

Assume that the creation of `newDiv` and buggy removal of that node by assignment to `innerHTML` is repeated a few times. How does one begin to notice a memory problem?

As discussed in Section 1, MemInsight reports which objects are *stale* and will not be used again, using an analysis for computing object lifetimes. Figure 4 shows screen captures from the MemInsight GUI. The time-line chart (A) shows the count of all objects with time (in blue) as well as all stale objects with time (in orange). The increasing count of stale objects is a cause for concern. The pie-chart on the right shows which allocation sites are the leading contributors to stale objects.

If we click on the pie chart slice for line 4 of `scr_orig.js`, MemInsight offers details for that allocation site, shown in Figure 4(C). We see a timeline chart for only the objects allocated from that site; a call tree leading up to that site (here it is called from main) and on demand, access paths to objects allocated at that site. The access paths show that the function object is held on by a `cache` internal to jQuery.

We illustrate a case of DOM leak as well. Going through the list of sites that MemInsight finds having a high count of stale objects, we come across the DOM nodes allocated at line 490 in jQuery. For a DOM node *d*, staleness means that
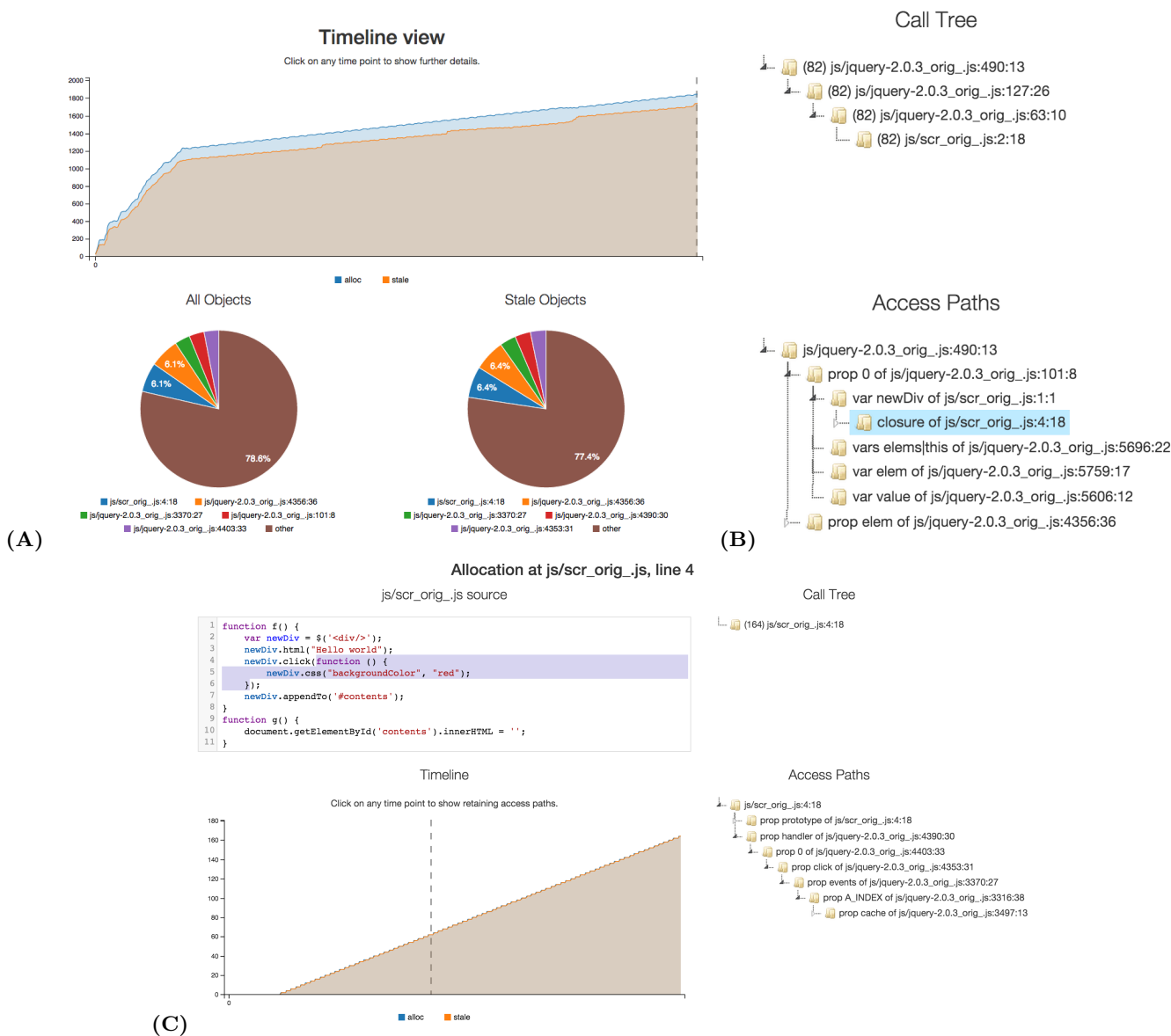
Figure 4: A,C. Memory profiling for the example in Fig 3. B. Call tree and access paths for jQuery line 490.

d is unused, detached from the document tree for the last time–hence not to be displayed again–and reachable from some JavaScript variable.

How does the user associate the allocation of DOM objects at line 490 in jQuery to the program in Figure 3? Details provided by MemInsight for line 490 show the call tree and access paths shown in Figure 4(B), which informs us that the allocation is the result of the `$` function called at line 2 in Figure 3. But why does it show increasing staleness? The access paths reveal the answer: the variable `newDiv` is in the closure of the handler allocated at line 4 in Figure 3. We previously saw why the latter was leaking. This now shows that, inadvertently, that causes a DOM node leak as well.

Incidentally, to discover DOM staleness, a tool must know the structure of the DOM *as it evolves*, noting when DOM nodes become *detached* from the main tree. This information is not evident from JavaScript execution alone: for this example, the nodes are allocated by calling the native method `createElement` and become detached from the main DOM due to the write to `innerHTML` at line 11 in Figure 3. Mem-Insight's DOM modeling techniques enable it to handle this case correctly.

3. TRACE GENERATION AND LIFETIME ANALYSIS

In this section, we detail the trace generation and lifetime analysis functionality of MemInsight. Basic trace generation, described in Section 3.1, uses source instrumentation to log a detailed trace of an execution's memory operations. The lifetime analysis, described in Section 3.2, uses this trace to compute exact object lifetimes, yielding an enhanced trace. Beyond this core functionality, we also describe MemInsight's carefully-designed handling of uninstrumented code in Section 3.3 and DOM APIs in Section 3.4.

```
1   var x = {};
2   var y = {};
3   function m(p,q)
4   {
5     p.f = q;
6   };
7   m(x,y);
8   x = null;
```

DECLARE x,y,m;
ALLOCOBJ 2 at 1;
WRITE x,2 at 1;
ALLOCOBJ 3 at 2;
WRITE y,3 at 2;
ALLOCFUN 4 at 3;
WRITE m,4 at 3;
CALL 4 at 7;
DECLARE p = 2,
        q = 3;
PUTFIELD 2,"f",3
        at 5;

LASTUSE 2 at 5;
RETURN at 7;
LASTUSE 4 at 7;
WRITE x,0 at 8;
UNREACHABLE
        2 at 8;
UNREACHABLE
        3 at end;
UNREACHABLE
        4 at end;

Figure 5: A simple code example and the corresponding trace. Red entries are added in the enhanced trace.

Finally, we discuss handling of various tricky JavaScript constructs in Section 3.5, and in Section 3.6 we describe key optimizations that help to reduce MEMINSIGHT's overhead.

3.1 Basic Trace Generation

Our generated traces must be sufficient to reconstruct object lifetimes, i.e., when objects are created and become unreachable. At a minimum, traces must include records of each object allocation and each memory write, both to variables and to object fields ("properties" in JavaScript parlance). (A JavaScript `delete` operation on an object, which removes a property, is modeled as a write of `null`.) To enable proper handling of JavaScript functions and closures in the lifetime analysis, the generator logs calls and returns, and also logs declarations of local variables. Finally, to enable reasoning about object staleness, we also log information about uses of each object: an object is used when it is dereferenced or, for function objects, when it is invoked.

By default, MEMINSIGHT enables two optimizations that reduce trace size and instrumentation overhead. First, we avoid logging writes when the old and new values are both of primitive type; such writes are irrelevant for reasoning about object staleness and leaks. Similarly, by default we only log the *last* use of each object, rather than all uses, since this information is sufficient for detecting staleness. These optimizations can be disabled at some cost in overhead (see Section 6), in case the additional information would be useful for some client analysis (e.g., object-equality profiling [21, 37]; further discussion in Section 7).

Figure 5 shows the generated trace for a simple example. Most entries includes a source location at the end. The allocation entries introduce a unique id used to name the corresponding object throughout the trace. We use a distinct entry type to identify function object allocation, to ease handling of closures in the lifetime analysis. In our implementation, LASTUSE entries include a timestamp and appear at the end of the generated trace (since the last use is only known at the end of the program); a separate post-processing phase inserts the entries at the appropriate slots.

3.2 Lifetime Analysis

Here we describe how our lifetime analysis computes exact object lifetimes based on the initial trace, inserting UNREACHABLE events into an enhanced trace, shown as the red events in Figure 5.

Heap graphs The trace as described in Section 3.1 enables simulation of the execution's heap graph. At each ALLOCOBJ and ALLOCFUN trace entry, a new vertex is added to the graph. At each PUTFIELD entry, an appropriate heap graph edge (labeled with the field name) is added and/or removed. Reasoning about variable writes in the presence of JavaScript closures is non-trivial, particularly since functions can update closure variables. For example:

```
function f() {
  var x = { p: 5 } /* o1 */, y = [1,2,3] /* o2 */;
  return { set: function (z) { x = z; },
           get: function () { return x; }
         }
}
var r = f(); r.set(null);
```

Here, `x` is present in the returned `set` and `get` closures (with updates via `set` reflected in future calls to `get`), but `y` is not, since it is not referenced by any nested function. Hence, the object labeled `o2` above becomes unreachable after the call to `f`, but `o1` remains reachable until after the call to `r.set`.

The heap graph construction handles closures using an approach similar to the "cactus stack" method of implementing first class continuations [6]. *Context nodes* in the heap graph represent variable bindings for closures, with parent pointers for lexical scoping. A separate syntactic analysis computes which variables from enclosing scopes are captured by each function, enabling a context to be *sealed* at function returns to remove pointers to uncaptured objects. For the example above, when the call to `f` completes, sealing removes the outgoing edge from the context corresponding to variable `y`. We correctly handle corner cases in JavaScript semantics, e.g., writing to an undeclared variable creates a fresh global.

Exact lifetimes Given our simulated heap graph, we use standard reference counting to compute exact unreachability times for all objects except those in cyclic data structures, which reference counting cannot handle. To handle cyclic structures, we adopt techniques from lazy cycle collection [19]. Whenever an object o's reference count is decremented to a non-zero value, o is placed in a cycle queue, as it may be in an unreachable cycle. To detect the true cycles, we periodically perform a mark-sweep over the heap, sweeping only from *unmarked* nodes in the cycle queue (sufficient since the base reference counter handles acyclic structures).

To detect exact unreachability times for nodes in cycles, we adopted the Merlin algorithm [14]. Whenever a node is placed in the cycle queue, it is stamped with the current time τ. If the node is already in the queue, its timestamp is updated. When sweeping unmarked queue nodes, the queue is first sorted by descending timestamp. After this sort, we are guaranteed that the node n in or reachable from an unreachable cycle with the latest timestamp t will be processed *first*, and that t is the true unreachability time for any unmarked node reachable from n. So, we simply propagate t as the unreachability time for all unmarked nodes encountered from n. The Merlin algorithm is presented in detail in Hertz et al. [14]; Appendix B contains an example of applying the technique.

3.3 Uninstrumented Code

MEMINSIGHT works robustly in the presence of uninstrumented JavaScript code or native code from the environment. In principle, uninstrumented code could arbitrarily mutate any memory locations to which it has access. Attempting

```
1   var elem = document.createElement("div");
2   elem.innerHTML = "<p><h1>Hello World!</h1></p>";
3   document.getElementById("x").appendChild(elem);
```

Figure 6: Example to illustrate handling of DOM-related code.

to discover all such behavior via code instrumentation alone would be difficult or impossible, particularly since invocations of uninstrumented code may not be observable (e.g., a browser invoking an uninstrumented event handler). Furthermore, such detection would require frequent reflective traversals of all heap state visible to uninstrumented code, a very costly operation.

In practice, we have found a policy of *only tracking references created in instrumented code* to strike a good balance between coverage of relevant behaviors and analysis overhead. The policy is simple: at any variable or field write in instrumented code, if the value is a hitherto unseen object–which must have been allocated by uninstrumented code–we treat the write as "allocating" the object. For example, consider line 1 in Figure 6, which creates a DOM node using the native `createElement` function. At the write to `elem`, our instrumentation emits log entries ALLOCOBJ i `at 1` and WRITE `y,`i `at 1` (i is a fresh object ID), modeling line 1 as "allocating" the DOM node.

Our analysis does *not* attempt to infer information about object references created in uninstrumented code. For the Figure 6 example, `document.createElement` is itself a function object allocated by uninstrumented code. However, since the function is only invoked, and not assigned to a variable or field, we do not assign it an object ID. While this makes our modeling less complete, we found that assigning object IDs in such cases added excessive noise to tool output and was not helpful in diagnosing memory issues in instrumented code. Unmodeled native code can cause unusual trace events, e.g., an object "re-appearing" after becoming unreachable due to an unknown pointer. Our enhanced trace generator (Section 3.2) detects and handles such issues, and can report them to the developer in case further modeling or instrumentation is desired.

MEMINSIGHT employs handwritten models for key built-in functions whose behavior is usually relevant to memory analyses. Modeled functions include the array-manipulating functions in `Array.prototype` (`push`, `pop`, etc.), `Object.defineProperty`, and `setTimeout`.

3.4 The DOM

Since the DOM is often involved in leaks and other bad memory usage patterns (see Section 2), we employ special handling to provide better visibility into its structure. A key difficulty in capturing DOM behavior via JavaScript instrumentation is handling the many ways the DOM may be mutated. Figure 6 shows two ways to mutate the DOM from JavaScript: assignment to `innerHTML` (line 2) and invoking the `appendChild` method (line 3). Many other JavaScript methods mutate the DOM, making fully-manual modeling arduous. Moreover, the construction of an initial DOM from static HTML occurs outside of JavaScript execution.

For a baseline handling of all methods of DOM mutation, MEMINSIGHT uses the *mutation observers* [24] functionality available in modern browsers. A mutation observer receives asynchronous notifications of mutation events for a set of observed DOM nodes. MEMINSIGHT attaches a mutation observer to as many DOM nodes as possible and use the notifications to reflect DOM mutations into the trace.

To attach its mutation observer, MEMINSIGHT performs an initial DOM traversal after the page loads, attaching the observer to each DOM node encountered. The observer then attaches itself to newly-inserted DOM nodes as it is notified of the insertions. DOM nodes are modeled as having a *set* of child nodes rather than an ordered list; this is sufficient for memory analyses, and it reduces analysis overhead and complexity.[1] DOM tree mutations are modeled with special ADDCHILD and REMOVECHILD log entries in the trace.

Let us see how this mutation observer technique works for the code in Figure 6. Assume that the mutation observer is already attached to the node with ID `x`, e.g., through the initial DOM traversal. After the code executes, the observer will be notified that the `elem` node has been added as a child of `x`, which it models with an ADDCHILD log entry. The observer then traverses the DOM sub-tree rooted at `elem`, discovering the nodes added at line 2; these nodes are reflected in the trace via appropriate ALLOCOBJ and ADDCHILD entries.

While the above technique enables accurate modeling of DOM structure, further modeling is sometimes required to obtain informative DOM node allocation sites. For the code of Figure 6, the nodes created by the `innerHTML` assignment on line 2 are no longer associated with that line by the time the (asynchronous) mutation observer runs. To address this problem, MEMINSIGHT performs an eager traversal of the DOM trees created by `innerHTML` and such constructs, recording an accurate location for the corresponding nodes.

Finally, MEMINSIGHT flags the root node of the main DOM tree in the trace. By knowing the root node and processing ADDCHILD and REMOVECHILD entries appropriately, client analyses can determine which nodes reside in the main DOM tree at any point. Such reasoning is very useful for tracking object staleness, as the analysis can avoid inaccurately treating "visible" DOM objects (those in the main DOM tree) as stale, though no uses may be observed for those nodes. We have found this functionality to be critical for reasoning about leaking DOM nodes in practice.

3.5 JavaScript Challenges

JavaScript's semantics have a number of corner cases that require careful handling during both trace generation and lifetime computation. Here, we briefly describe MEMINSIGHT's handling of some interesting language features.

Normalization We built our trace generator atop the JALANGI framework [32], which does not normalize code to a three-address form before instrumentation. Hence, sub-expressions must be handled carefully, to avoid marking an object as unreachable too soon. Consider the simple expression `f({a:1}, {b:2})`, which allocates two objects and passes them to `f`. For this expression, JALANGI provides three callbacks, first for the `{a:1}` literal, then for `{b:2}`, and finally for the call. In the first callback, the analysis observes no reference being created to the `{a:1}` literal, but it clearly should not be marked unreachable at this point. This issue could of course be addressed by first normalizing the source code. But, this would cause extra code bloat, which could further slow execution. Also, normalization could change

---

[1]Modeling child ordering is tricky since a single mutation could change the position of many other child nodes.

object lifetimes if temporary variables were not carefully nulled out after use, which would introduce even more bloat. Instead, the trace generator issues FLUSH entries at the end of each source-level statement (e.g., after the call to `f` for the example above), and the lifetime analysis only marks nodes as unreachable at a FLUSH.

Constructors The complex semantics of JavaScript constructor calls require special handling. One key complication is that *any* JavaScript function f can be invoked as a constructor via the `new` operator. If f has no explicit return statement, the value of `new f(...)` is a fresh object passed as `this` to f (see the example below). However, if f returns some other object o, then o becomes the value of the `new` expression. We employ special logic to ensure the `this` value is flagged as the return value in the former case, so it does not appear to become unreachable at the end of the constructor call.

A second issue is recording a good source location for constructor-allocated objects. Consider this example:

```
1  function C() { this.f = 3; }
2  var x = new C();
```

Line 2 should be associated with the new `C` object, but at the source level, the new object cannot be observed until execution of the `C` call has already begun. To handle this common case, our implementation assigns an ID to the `this` object at the beginning of a constructor call, and it flags the object as "under construction." When an under-construction object is detected as the value of a completed constructor call, we clear the flag and log an UPDATELOC entry to update the allocation site for the object. For the above example, the `C` object would be allocated some ID i at the beginning of the `C` function, and the log would later include an UPDATELOC `i at 2` entry to set its allocation site as line 2.

Other features Various other JavaScript features require special handling; we list a few cases here. For prototype chains, we model the creation of the `__proto__` property at constructor calls, and the `prototype` property of functions. We reserve object ID 1 for the JavaScript global object [12] (whose properties are the global variables), as code can reference and create aliases to this object. Getters and setters [26], which respectively enable accessor functions to execute at property reads and writes, require care. In particular, before logging a field write, we must ensure that the field is not a setter, since in this case the write expression invokes the setter. Finally, JavaScript's `eval` construct is handled fully (including indirect `eval`, which has different scoping semantics) by using JALANGI to instrument the code before evaluation.

3.6 Optimizations

Here, we discuss some of the key optimizations we implemented to make MEMINSIGHT scale; these techniques may apply to other JavaScript dynamic analyses.

Scaling the Lifetime Analysis In principle, one could implement MEMINSIGHT's lifetime analysis *in situ* with a running application via code instrumentation—the requisite heap traversals can be achieved using JavaScript's reflective constructs (iteration over object properties and dynamic property accesses). However, this approach would induce such high overheads as to make most applications unusable. To address such cases, JALANGI provides a record-replay framework [32] that allows for an expensive dynamic analysis to be executed during an offline replay of the application, with the analysis running in situ with the replayed application to ease implementation.

We first implemented our object lifetime analysis using JALANGI replay, but we found that running the analysis in situ did not scale, even during offline replay. A key problem was that JavaScript's reflective property access constructs have not been highly optimized in JavaScript runtimes, making reflective heap traversals very expensive. Furthermore, implementing the analysis in situ required careful handling of both prototype chains and getters and setters during traversals, which was both fragile and further slowed the analysis. Finally, our custom DOM modeling (Section 3.3) did not fit well with JALANGI record-replay, as it required persisting additional information during record mode.

To avoid these issues, we switched to the approach of generating custom memory traces and running the lifetime analysis over a simulated heap graph, which led to orders-of-magnitude speed improvements. MEMINSIGHT runs the lifetime analysis in a concurrent process as the memory trace is generated by the instrumented application, so in many cases, the lifetime analysis requires no additional running time. Also, having full control of the trace format greatly simplified modeling of DOM operations.

Scaling Trace Generation We implemented a number of techniques to reduce the overhead of trace generation in MEMINSIGHT. Since we were not using JALANGI's record-replay feature, we implemented a specialized analysis runtime in JALANGI without record support, reducing JALANGI's baseline overhead. We also enhanced JALANGI to support optional disabling of instrumentation for certain language constructs, and leveraged this feature to completely avoid instrumenting operations irrelevant to MEMINSIGHT (arithmetic operations, etc.).

Within our trace generation code, we encoded each object's metadata in a single 32-bit value to avoid unnecessary heap allocation. For maintaining the per-object metadata, we implemented two different strategies, one that stored metadata in a hidden object property and one that kept metadata in a `WeakMap`, a recently-added JavaScript feature.[2] We found that while time overhead was mostly comparable for the two approaches, hidden object properties caused a space blowup in some cases (see Section 6), likely due to some bad interaction with the JavaScript runtime. Hence, MEMINSIGHT uses a `WeakMap` when available. Finally, we write traces in a compact binary format, minimizing the amount of overhead required to create trace entries.

4. CLIENT ANALYSES

Based on the enhanced trace produced by the lifetime analysis (see Figure 2), we have implemented a number of useful client analyses to expose possible memory issues in the trace. As illustrated previously in Figure 1, we compute detailed information on object staleness to expose drag-related issues. We also implemented a client that can compute the access paths retaining some set of objects at some time, as shown in Figure 4(B). Below we discuss some other useful memory-related properties that we compute for each allocation site in the program. A developer could easily compute other interesting memory analyses using the enhanced trace (see Section 7 for some discussion).

---

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap`

Let an 'ℓ-object' be an object allocated at site ℓ. For each such site ℓ, we compute the following information:

- **isUnused**: this flag is set if none of the ℓ-objects are ever used, indicating that the ℓ-objects may be unnecessary.
- **isOneAliveAtATime**: this flag is set if at most one ℓ-object is *alive* (i.e., reachable) at any state during the execution. If this holds, it may be possible to replace the allocations with a single global object, reducing stress on the garbage collector.
- **isNonEscaping**: this flag is set if none of the ℓ-objects escape their allocating function, indicating an opportunity for stack allocation of the corresponding data.
- **isLeaking**: the flag is set to true if the objects allocated at ℓ exhibit a behavior indicative of a leak.
- **consistentlyPointedBy**: this property is set to site $\ell' \neq \ell$ if each ℓ-object is uniquely pointed-to throughout its lifetime by an ℓ'-object. In such a scenario, one may be able to *inline* the properties of ℓ-objects into the corresponding ℓ'-objects [9]. This optimization can boost performance by reducing allocations and the number of dereferences required to access ℓ-object properties.

Given an enhanced trace containing unreachability events, the computation of the boolean flags `isUnused` and `isOneAliveAtATime` is straightforward. We experimented with a number of techniques for computing the `isLeaking` flag, and settled on the following criterion: `isLeaking` for ℓ is set to true if, at all consecutive execution points where the call stack is empty, the number of stale objects allocated from ℓ is *strictly increasing*. We only consider points when the call stack is empty since this represents an idle state in a web application (e.g., between event handlers), where temporary objects should have been released. While this technique can miss leaks, we found that techniques that tolerated decreases in stale objects led to too many false positive leak reports.

To compute the remaining two properties, we use an *execution index* data structure, as described by Xin et al. [36]. The execution index of a program state s is a sequence of the form $[(\ell_1, q_1), (\ell_2, q_2), \ldots, (\ell_n, q_n)]$, where each ℓ_i is a call site on the call stack for s, and q_i is the execution count for ℓ_i in s. By including both the call stack and execution counts for each contained call site, an execution index uniquely identifies a function call in an execution. To compute the `isNonEscaping` property, we exploit the fact that for any execution indices e_1 and e_2, e_1 is a prefix of e_2 iff e_2 is the execution index of a state reached after calling 0 or more functions in the dynamic context denoted by e_1. We associate with each object o the execution index e_a of the state in which the object is created and index e_u of the state where it becomes unreachable; o escapes if e_a is not a prefix of e_u. If no ℓ-object escapes, then ℓ is `isNonEscaping`.

We compute the `consistentlyPointedBy` flag as follows. We associate a meta-property `pointedBy` with each object o, initialized to $\perp$ when o is created. After creation, if o is then written into a property of object o' via a PUTFIELD operation, we set o's `pointedBy` to o'. If o is *ever* referenced by any other variable or object property, we set its `pointedBy` to $\top$. The `consistentlyPointedBy` property is set to ℓ' if the `pointedBy` meta-property of each ℓ-object is an ℓ'-object, *and* if each paired ℓ and ℓ'-object have similar lifetimes. We check for similar lifetimes for two objects by ensuring that (1) they are both allocated under the same execution index and (2) they both become unreachable at the same time.

This lifetime check ensures that the parent object "owns" the child throughout its lifetime. If `consistentlyPointedBy` is set to ℓ' for some ℓ, then the ℓ objects are a good candidate for object inlining into the ℓ' objects.

5. CASE STUDIES

Among the web and `node.js` applications that have been tested using MEMINSIGHT, memory issues have been discovered or confirmed in ten cases, with several issues found by a separate product team using the tool. In three cases, patches were submitted to the developers and accepted. In this section, we highlight some of these memory issues in more detail. (None of the issues reported by the product team are discussed here.)

shopping list Shopping List [35] is a Tizen web application. For this app, MEMINSIGHT showed leaking objects and DOM nodes deep inside the jQuery library. Call stacks at the allocation points of the leaking objects showed that they were allocated due to application calls for attaching event handlers. Access paths showed that the objects remained reachable due to pointers from an internal jQuery cache.

The cause of the leak was similar to the defect shown in Figure 3. The code attempted to clear a DOM element as follows:

```
if (self.currentView.resetListOfLists) {
  ShoppingListApp.listoflists.innerHTML = "";
}
```

The correct way when using jQuery would be to use jQuery construct `$(ShoppingListApp.listoflists).empty();` in place of an assignment to `innerHTML`. This changes fixed the leak, and a patch was accepted by the developer.[3]

annex The annex app was discussed previously in Section 1. We exercised annex by playing in 1-person mode (the computer responding to human moves) for three moves. MEMINSIGHT identified excessive drag, as shown previously in Figure 1. Most stale objects were being allocated within a function used for move evaluation. MEMINSIGHT's access path view for the site revealed that these objects were accumulated in a global recursive data structure corresponding to computing a minimax game tree. The code uses the following computation to identify the best (highest-value) computer move from possible moves:

```
ret = possible[0];
var value = this.evaluate(ret);
for (var p=1; p<possible.length; p++) {
    var v = this.evaluate(possible[p]);
    if (v > value){
        value = v;
        ret = possible[p];
    }
}
```

The same game tree is retained across multiple calls to `evaluate`, but the code could equivalently start with a fresh tree every time, freeing stale memory earlier. We applied this refactoring and found a drastic reduction in the peak counts as well as staleness of objects.[4] We have contacted the developer, who has agreed to look at the issue (though has not confirmed our fix as of this writing).

---

[3]See `http://goo.gl/Xx5eRo`.

[4]The timeline graph for the fixed app appears in Appendix C.

```
return {                        return {
  type: type,                     type: type,
  value: id,                      value: id,
  lineNumber: lineNumber,         lineNumber: lineNumber,
  lineStart: lineStart,           lineStart: lineStart,
  range: [start, index]           start: start,
};                                end: index
                                };
```

Figure 7: An excerpt from esprima illustrating bloat and object inlining

MEMINSIGHT also identified an allocation site with high count and the `isNonEscaping` and `isOneAliveAtATime` flags set (see Section 4) in the following code (edited for brevity):[5]

```
getValue: function(place){
  var i = parseInt(place[0]);
  var j = parseInt(place[1]);
  var mtable = {
    0:{ 0:100, 1:-50, 2:40, 3:30, 4:30, 5:40, ...},
    1:{ 0:-50, 1:-30, 2:5, 3:1, 4:1, 5:5, ...},
    ...
  };
  return parseInt(mtable[i][j]);
}
```

The object assigned to `mtable` was identical for each call to `getValue`, and these allocations were causing significant unnecessary churn. The issues could be fixed by replacing the allocations with a single global object. Performing this transformation led to a 10% bottom-line speedup for computer move computation, and the fix was accepted by the developer.[6]

dataTables DataTables [7] is a popular JavaScript library that, among other things, provides a way to render a large table in paginated form, e.g. ten rows at a time with navigation support. We exercised DataTables to reproduce a previously-identified leak of DOM nodes [8], by loading a table, navigating through a few pages, and then destroying the table. MEMINSIGHT indicated excessive staleness at the site allocating the DOM nodes, and the access paths view showed a settings object on all paths retaining the nodes. We applied a suggested patch that nulls out the contents of the settings object [8], and confirmed that the problem was fixed. Note that MEMINSIGHT's support for reasoning about visible DOM nodes was critical for identifying the truly stale, leaking nodes.

esprima Esprima [11] is a widely-used JavaScript parser written in JavaScript. For our case study, we used an older version of Esprima with code like the left fragment in Figure 7. In this code, the object literal being returned points to an array of length two throughout its lifetime via the `range` field. As this array is always accessed through the returned object, we can reduce bloat in the data structure by inlining this child array into the object. We ran MEMINSIGHT on the unfixed version, and the `consistentlyPointedBy` flag described in Section 4 flagged the site as a candidate for object inlining. A subsequent commit by the developers inlined the objects as in the right code fragment of Figure 7,[7]

yielding a roughly 10% bottom-line speedup when parsing large JavaScript files. Note that knowledge of object lifetimes was key to determining that these objects were inlineable.

escodegen Escodegen [10] serializes JavaScript ASTs back to source code. We noticed poor performance in Escodegen for large ASTs, so we investigated by running Escodegen with MEMINSIGHT for a smaller input. MEMINSIGHT immediately identified a very high allocation count of `SourceNodeMock` objects. However, MEMINSIGHT did not show a leak or excessive staleness for the objects. Further manual investigation showed that `SourceNodeMock` objects were unnecessary in the common case, and a patch eliminating the objects yielded a roughly 10X speedup for large inputs and was accepted by the developers.[8] While MEMINSIGHT could not automatically identify this complex refactoring opportunity, it still provided useful information, in particular that the objects were not leaking and that a deeper fix would likely be needed.

6. EVALUATION OF IMPLEMENTATION

MEMINSIGHT is implemented primarliy in TypeScript, with trace generation implemented using the JALANGI framework [32]. The lifetime analysis is implemented in Java for greater efficiency and to leverage existing libraries. In addition to the case studies discussed in Section 5, we measured MEMINSIGHT's effectiveness according to the following two evaluation criteria:

(EC1) Does MEMINSIGHT run with reasonable overhead for real-world, interactive web applications?

(EC2) Does MEMINSIGHT effectively expose DOM-related memory behavior?

6.1 Overhead

In our experience running with several real-world applications, we found that the slowdown introduced by MEMINSIGHT due to trace generation was reasonably low, with the applications remaining usable and interactive. Since interactive applications tend not to be very compute-intensive, our JavaScript instrumentation does not dramatically impact their usability. As rigorous measurement of overhead on interactive applications is difficult, we also ran MEMINSIGHT on programs from the Octane benchmark suite [28],[9] and found that MEMINSIGHT introduced a mean of 41.7X time overhead (11.8X–102.3X).[10] This overhead is comparable with previous instrumentation tools and analyses [14, 20, 25, 30], and for real-world web applications, this overhead only applies to JavaScript execution, typically a small percentage of total execution time.

We also measured the overhead of MEMINSIGHT on Octane while disabling the two optimizations described in Section 3.1, namely recording of only the last use of an object and eliding putfields where the value is a primitive type. We found that recording all uses had a mean overhead of 55.1X, and recording all putfields had a mean 47.9X overhead. While somewhat higher than the default configuration, these overheads are still manageable and may be worth incurring to

---

[5]This allocation site was also noted in previous work [32], by a simpler analysis that only tracked allocation counts; it did not track whether allocated objects escaped.

[6]See http://goo.gl/hWfIYm.

[7]See http://goo.gl/I3wLs8.

[8]See http://goo.gl/XlBX51.

[9]We excluded the `mandreel` benchmark as, at roughly 277 KLOC (automatically generated from a large C++ program), it exceeded the size that could be instrumented by JALANGI.

[10]For a detailed table of all experimental data, see Appendix A.

Table 1: DOM node measurements

app	base	model	init	unknown	det
annex	45	290	215	0	46
shopping	604	1298	993	0	1592
dataTables	607	233	47	0	768
hackerweb	8	1850	185	14	872

enable other compelling clients (see Section 7 for further discussion). Finally, we found that our lifetime analysis ran in an average of 3.9 seconds on the Octane benchmarks (0.4s–162.4s). As noted in Section 3.6, MEMINSIGHT runs this phase concurrently with trace generation, and for typical web applications this phase requires no extra time.

6.2 DOM Handling

Table 1 quantifies the effectiveness of MEMINSIGHT's DOM modeling for the three web applications used as case studies (to be described further in Section 5), and also hackerweb [13], an additional app that does significant DOM mutation. For each application, the table gives the number of DOM nodes with allocation sites from our baseline handling of uninstrumented code ("base"), nodes with allocation sites from the additional DOM modeling of Section 3.3 ("model"), nodes in the initial DOM from parsing static HTML ("init"), and nodes with unknown allocation sites ("unknown"). Overall, MEMINSIGHT's DOM modeling was very effective, exposing many more DOM allocation sites via modeling as compared to the baseline. The "det" column gives the number of nodes observed to be detached from the main DOM tree during execution. Such nodes must be carefully managed to avoid leaks, and the significant number of detached nodes in the apps shows the property's importance for a memory profiler.

7. RELATED WORK

Object lifetime analyses Our enhanced trace format has similarities to garbage collection traces [14], used to simulate garbage collection algorithms. We adopted the Merlin algorithm [14] to compute exact object lifetimes (see Section 3.2). Later work by Ricci et al. [30] enhances the GC trace format in a fashion similar to ours with call/return information. Resurrector [38] is a lifetime analysis technique which incurs much less overhead than Merlin while maintaining mostly-precise lifetimes. Resurrector leverages a modified virtual machine, so we cannot easily adopt its techniques.

Biographical profiling In the context of functional languages executed by graph reduction, Röjemo and Runciman [31] present a profiler that measures lag, drag (staleness) and void. Lag is the time from creation until first use, and void is an object that is never used. All of these metrics can be computed using MEMINSIGHT, with lag requiring tracing of the first and last use of each object.

Leak detection One application of MEMINSIGHT is leak detection for JavaScript programs. Much previous work on leak detection exists; we will cover the most pertinent here. AjaxScope [17] includes a leak detector specifically aimed at finding heap cycles involving DOM nodes, which could not be garbage collected in older browsers. JSWhiz [29] is a static analysis that detects memory leaks caused by misuses of the Google Closure framework.

Chilimbi and Hauswirth present the SWAT tool [4] which uses statistical profiling to predict if an object is stale or will be used again, based on its previous access patterns. In Sleigh [3], Bond and McKinley adapt the SWAT approach to Java and reduce the overhead by using unused bits in the object header. MEMINSIGHT computes staleness precisely without modifying the runtime, but with higher overhead than these approaches. Shaham et al. [33, 34] use an instrumented VM to find stale objects and the references keeping it alive. Furthermore the tool computes when a programmer can safely remove a reference to reduce staleness. MEMINSIGHT enables a programmer to do the same by providing access paths and last use sites.

With LeakBot [23] Mitchell and Sevitsky detect leaks in enterprise Java applications by identifying memory regions that grow unboundedly, based on initial heap snapshots and selective tracing. A similar approach is taken in Cork [16], using summarization instead of snapshots. Detecting growing regions is complementary to staleness as currently computed by MEMINSIGHT, and we leave this to future work.

GC assertion frameworks [1] allow programmers to write assertions about object lifetimes and then verify these assertions based on a garbage collection trace. LeakChaser [40] allows the programmer to assert invariants about object lifetimes and to define transaction boundaries in the program that correspond to object lifetimes. Mitchell [22] uses heap dominators to present a view of a memory use that allows a programmer to see which objects are responsible for large subgraphs. As future work we will consider similar ways to present results to the programmer.

Bloat Reduction Many techniques have been proposed for helping developers detect run-time bloat. Copy profiling [39] and the more general reference propagation profiling [41] track data flow to expose issues like excessive allocation of temporary objects. Object-equality profiling [21, 37] explores another way of reducing bloat, in which sets of equivalent objects are replaced with a single representative object. Cachetor [27] is able to find repeated allocation of identical objects. Techniques similar to these could be implemented atop MEMINSIGHT when configured to record all reads and writes to the heap; we plan to investigate such techniques in future work.

Object inlining is a well-studied method for reducing bloat [9]. Lhoták and Hendren [18] studied possibilities of object inlining in Java applications via tracing. They also performed a finer-grained inlining analysis, where individual fields are considered for inlining into the parent instead of only entire objects. This finer-grained analysis could also be built on MEMINSIGHT.

8. LIMITATIONS

Being a dynamic analysis technique, MEMINSIGHT can miss a real memory issue if it does not happen during a test execution. Moreover, MEMINSIGHT may not be able to detect a memory issue if the application is run for a short period of time (e.g. a single move in the annex game is not sufficient to expose the memory drag problem.) This limitation can be removed by testing a web application more exhaustively. Also, MEMINSIGHT uses heuristics to detect memory leaks and drags. We developed these heuristics based on existing research results and our experience with several web apps. In case of drag or leak reports, a dynamic analysis based on a single execution cannot conclusively infer that a stale or a leaky object is not going to be used in future. Therefore, MEMINSIGHT can give false warnings.

9. CONCLUSIONS

We have presented MEMINSIGHT, a platform-independent tool for detailed memory profiling of JavaScript applications. On unmodified JavaScript runtimes, MEMINSIGHT can handle complex JavaScript constructs, expose memory behavior of DOM objects, and compute exact object lifetimes. Our experimental results show that these rich features can be achieved with reasonable runtime overhead, and that MEMINSIGHT's functionality can expose memory issues in real-world applications.

References

[1] E. Aftandilian and S. Z. Guyer. GC assertions: using the garbage collector to check heap properties. In *PLDI*, pages 235–244, 2009.

[2] HTML5WebApps - Annex. `https://01.org/html5webapps/webapps/annex`. Accessed: 2015-07-14.

[3] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.

[4] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.

[5] Chrome developer tools overview. `https://developer.chrome.com/devtools/index`. Accessed: 2015-07-14.

[6] W. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 124–131, New York, NY, USA, 1988. ACM.

[7] Datatables - table plug-in for jQuery. `http://datatables.net/`. Accessed: 2015-07-14.

[8] Datatables patch-1. `http://www.mozartrocks.ro/dt-test/patch-1.html`. Accessed: 2015-07-14.

[9] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, 2000.

[10] Escodegen. `https://github.com/Constellation/escodegen`. Accessed: 2015-07-14.

[11] Esprima. `http://esprima.org`. Accessed: 2015-07-14.

[12] ECMAScript language specification - ECMA-262 edition 5.1: The global object. `http://www.ecma-international.org/ecma-262/5.1/#sec-15.1`. Accessed: 2015-07-14.

[13] Hackerweb – a simply readable hacker news web app. `http://hackerwebapp.com/`. Accessed: 2015-07-14.

[14] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Generating object lifetime traces with Merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, 2006.

[15] jQuery. `http://jquery.com`. Accessed: 2015-07-14.

[16] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.

[17] E. Kiciman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. *ACM Trans. Web*, 4(4):13:1–13:52, Sept. 2010.

[18] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005.

[19] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, 1992.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[21] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, 2003.

[22] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.

[23] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, pages 351–377, 2003.

[24] DOM standard: Mutation observers. `http://dom.spec.whatwg.org/#mutation-observers`. Accessed: 2015-07-14.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[26] M. D. Network. Working with objects: Defining getters and setters. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_getters_and_setters`. Accessed: 2015-07-14.

[27] K. Nguyen and G. H. Xu. Cachetor: detecting cacheable data to remove bloat. In *ESEC/FSE*, 2013.

[28] Octane. `https://developers.google.com/octane/`. Accessed: 2015-07-14.

[29] J. Piennar and R. Hundt. JSWhiz - static analysis for JavaScript memory leaks. In *Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization*, 2013.

[30] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: portable production of complete and precise GC traces. In *ISMM*, pages 109–118, 2013.

[31] N. Röjemo and C. Runciman. Lag, drag, void and use; heap profiling and space-efficient compilation revisited. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 34–41, New York, NY, USA, 1996. ACM.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*, 2013.
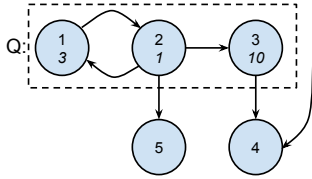
Figure 8: Cycle queue example. The vertices inside the dashed lines are in Q, the number in cursive is the associated timestamp.

[33] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *ISMM*, 2000.

[34] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *PLDI*, 2001.

[35] HTML5WebApps - Shopping list. `https://01.org/html5webapps/webapps/shopping-list`. Accessed: 2015-07-14.

[36] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, pages 238–248, 2008.

[37] G. H. Xu. Finding reusable data structures. In *OOPSLA*, 2012.

[38] G. H. Xu. Resurrector: a tunable object lifetime profiling technique for optimizing real-world programs. In *OOPSLA*, pages 111–130, 2013.

[39] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

[40] G. H. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. In *PLDI*, pages 270–282, 2011.

[41] D. Yan, G. H. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *ICSE*, 2012.

APPENDIX

A.   DETAILED RESULTS

Table 2 presents the data from our experiments applying MemInsight to a variety of benchmarks. In addition to the Octane benchmarks, we include escodegen and esprima, two node.js applications discussed in Section 5.

B.   LIFETIME ANALYSIS EXAMPLE

Here, we present an example of using the Merlin algorithm [14] to compute exact object lifetimes in the presence of heap cycles, as discussed in Section 3.2. Figure 8 illustrates part of a heap. Vertices 1 to 3 are in the cycle queue Q with time stamps, 3, 1 and 10 respectively. Vertices 1 and 2 are in an unreachable cycle, and vertices 5 and 3 are only reachable from this cycle. Vertex 4 is still reachable from some root. After flushing Q, the algorithm should compute unreachability time 3 for vertices 1, 2 and 5, and time 10 for vertex 3.

Let us step through the flush operation for the heap and Q as pictured. First, all vertices reachable from live contexts are marked, which here will only mark vertex 4. We now sort Q in descending order by time stamp, yielding order $3, 1, 2$. The algorithm now traverses unmarked nodes reachable from these vertices in turn. For vertex 3, only the vertex itself is found, since vertex 4 is already marked. Vertex 3 is recorded as unreachable at time 10, marked, and removed. Next is vertex 1, and in the traversal we find vertices 2 and 5 (but not 3, as it was marked in the previous iteration) and record their unreachability time as 3. Finally vertex 2 is already marked, so no traversal is performed.

C.   ANNEX, FIXED FOR DRAG ISSUE

Figure 9 shows MemInsight's timeline graph for annex after our fix for the drag issue discussed in Section 1.

bench	LOC	UT	UM	MT	AUT	APT	WMM	HPM	TS	ET
richards	300	0.05	1.0	11.8X	14.3X	12.3X	6.9X	7.0X	3.7	1.5
deltablue	476	0.05	1.0	15.2X	17.4X	15.8X	7.5X	8.4X	7.0	2.3
crypto	1306	0.06	1.0	51.0X	87.3X	67.6X	7.9X	8.2X	24.8	3.4
raytrace	682	0.06	2.6	42.9X	47.7X	42.6X	7.9X	4.9X	27.6	4.4
earley-boyer	3483	0.08	5.3	100.2X	110.7X	100.3X	7.8X	17.7X	145.1	73.7
regexp	1755	0.13	7.3	26.5X	32.3X	27.3X	4.3X	3.1X	32.9	5.0
splay	250	0.25	49.8	42.3X	44.4X	42.9X	2.7X	13.5X	102.0	20.8
navier-stokes	360	0.06	4.5	41.8X	76.3X	65.3X	2.7X	2.7X	6.1	0.6
pdfjs	29843	0.36	22.0	29.8X	36.6X	34.1X	3.8X	4.0X	166.1	9.2
gbemu	9482	0.26	9.1	90.5X	136.8X	117.0X	3.8X	3.9X	122.4	16.2
code-load	1422	0.06	3.4	36.3X	35.8X	36.2X	17.7X	17.7X	0.1	0.4
box2d	517	0.23	6.2	33.8X	50.5X	38.6X	4.4X	4.9X	57.7	8.8
zlib	2496	1.50	5.8	102.3X	216.9X	169.6X	13.1X	13.3X	234.7	10.7
typescript	23854	1.68	64.0	76.3X	96.1X	82.4X	2.5X	6.0X	1456.7	162.4
escodegen	1855	0.17	4.8	7.5X	8.4X	7.2X	4.3X	6.0X	13.2	5.1
esprima	3063	0.10	3.1	8.7X	9.7X	8.8X	4.4X	7.2X	7.8	2.9

Table 2: Overheads for running on Octane benchmarks and node.js case studies. The columns from left to right are: lines of code (LOC), uninstrumented running time in seconds (UT), uninstrumented memory usage in megabytes (UM), time overhead with minimal logging (MT), logging of all uses (AUT), and logging of all putfields (APT), memory overhead using WeakMaps (WMM) and hidden properties (HPM), trace size in megabytes (TS) and time taken to produce the enhanced trace in seconds (ET).
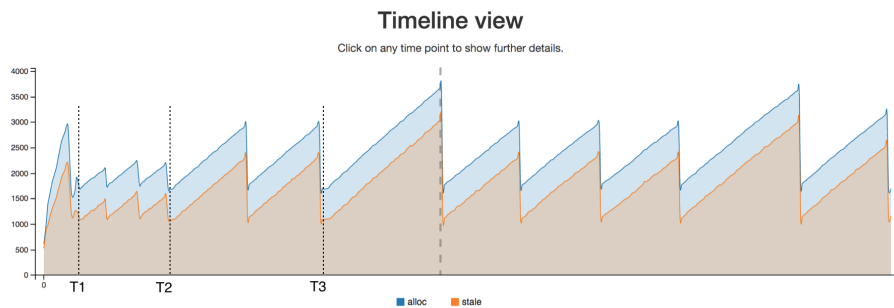
Figure 9: Number of objects in memory for three moves in the fixed version of *annex*, as shown by Mem-Insight. Note the peak number of objects is much less than in the original version, 4000 vs 10000. (See Figure 1)