FlashFreeze: Low-Overhead JavaScript Instrumentation for Function Serialization

Jonathan Van der Cruysse
Ghent University
Ghent, Belgium
jonathan.vdc@outlook.com

Lode Hoste
Nokia Bell Labs
Antwerp, Belgium
lode.hoste@nokia-bell-labs.com

Wolfgang Van Raemdonck
Nokia Bell Labs
Antwerp, Belgium
wolfgang.van_raemdonck@nokia-bell-labs.com

Abstract

Object serialization is important to a variety of applications, including session migration and distributed computing. A general JavaScript object serializer must support function serialization as functions are first-class objects. However, JavaScript offers no built-in function serialization and limits custom serializers by exposing no meta operator to query a function's captured variables. Code instrumentation can expose captured variables but state-of-the-art instrumentation techniques introduce high overheads, vary in supported syntax and/or use complex (de)serialization algorithms.

We introduce FlashFreeze, an instrumentation technique based on capture lists. FlashFreeze achieves a tiny run time overhead: an Octane score reduction of 3% compared to 76% for the state-of-the-art ThingsMigrate tool and 1% for the work-in-progress FSM tool. FlashFreeze supports all self-contained ECMAScript 5 programs except for specific uses of `eval`, `with`, and source code inspection. FlashFreeze's construction gives rise to simple (de)serialization algorithms.

CCS Concepts • **Software and its engineering → Preprocessors**; *Procedures, functions and subroutines*; *Runtime environments.*

Keywords JavaScript, TypeScript, Closures, Serialization, Instrumentation, Compilers

ACM Reference Format:
Jonathan Van der Cruysse, Lode Hoste, and Wolfgang Van Raemdonck. 2019. FlashFreeze: Low-Overhead JavaScript Instrumentation for Function Serialization. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (META '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3358502.3361268

1 Introduction

Serialization, the practice of encoding in-memory objects in such a way that they can be saved, transmitted and restored, is an invaluable tool for a range of applications including web session migration and distributed computing [10, 20].

The main challenge in serializing JavaScript objects is that of function serialization: JavaScript functions are first-class objects. Hence, any serializer for arbitrary objects must by definition be able to serialize functions. JavaScript functions consist of a function definition and a set of captured variables. A function's definition can be queried by a serializer using an API from the ECMAScript standard [9] for JavaScript, but there is no analogous API for querying captured variables.

State-of-the-art JavaScript serialization techniques either rewrite the JavaScript source code that defines the function to serialize or modify the virtual machine (VM) in which it executes [20, 22]. In both cases, the modifications make captured variable sets accessible to serializers at run time.

The source code rewriting approach—also known as *instrumentation*—is particularly convenient because it also applies in situations where using a modified VM is not an option, as is the case with, e.g., an end user–installed web browser.

However, state-of-the-art instrumentation-based JavaScript serializers incur considerable size and run time overhead compared to unmodified code. We refer to this penalty as *static overhead.* In this paper, we focus on minimizing static overhead. To that end, we make the following contributions:

- We introduce the novel FlashFreeze instrumentation strategy in section 3. The strategy is based on the notions of capture lists and by-value capture. It minimizes static overhead by taking a lazy approach to captured variable retrieval.
- In section 4, we compare FlashFreeze's design choices with those of other JavaScript function serializers.
- We quantify FlashFreeze's static overhead in section 5 using the Octane benchmark suite. For comparison, we do the same for ThingsMigrate [10] and FSM [18].

2 Motivation

Previous work indicates that there exists a large class of JavaScript applications for which arbitrary object serialization is a natural if not indispensable component [10, 19, 20, 22].

The seminal Imagen project [20] appears to be the first JavaScript function serializer that preserves lexical scopes. Imagen emerged as a means to take *application snapshots:* serialized versions of a web page's state that can be transferred and restored. To create snapshots, Imagen treats the state of a web page as an object graph to serialize. Such a graph may include function objects, which Imagen can serialize after first rewriting the web page's JavaScript source code.

Application snapshot creation has since sparked the appearance of other instrumentation-based serializers such as Fast Snaphot Migration (FSM) [18] as well as completely different approaches to JavaScript function serialization, including the use of modified JavaScript VMs [19, 22].

The ThingsMigrate project likewise implements arbitrary object graph serialization, but for a different, emerging use case: distributed computing [10]. ThingsMigrate is designed to transparently transfer stateful applications from one Internet of Things (IoT) device to another, as necessitated by, e.g., resource constraints. To transfer an application, ThingsMigrate creates a snapshot of the application's current state, transmits that snapshot, and restores it on another device.

A related scenario shows up in the World Wide Streams (WWS) distributed computing platform [27]. A master node serializes a function along with its dependencies and transmits it to a number of worker nodes. The workers deserialize the function with proper scope context and execute it.

JavaScript is not the only language for which distributed computing has motivated the emergence of function serialization. For example, Distributed Smalltalk can migrate objects along with the classes describing their behavior [4]. Similarly, Apache Ignite for Java supports *peer class loading*: when an object is moved between hosts, the bytecode describing that object's class is transferred as well [6]. Likewise, PyWren is a function-as-a-service (FaaS) library that serializes Python functions to execute them in the cloud [16].

However, JavaScript function serialization warrants special treatment because captured variables cannot be inspected in JavaScript. This is unlike many other contemporary programming languages such as Python and Java, which do allow serializers to inspect captured variables.

2.1 Limitations of JSON

JavaScript runtimes include native support for serialization and deserialization using JSON. However, these serialization and deserialization routines have a number of fundamental limitations that make them incapable of directly serializing arbitrary object graphs:

1. JSON serialization routines turn object graphs into object trees, resulting in object duplication on deserialization.
2. JSON serialization does not support cyclic object graphs.
3. JSON cannot serialize certain types of objects including functions.

```
function add(x, y) { return x + y; }
function twice(x) { return add(x, x); }
```

Listing 1. Simple variable capture: function `twice` captures `add`.

```
function add(x, y) { return x + y; }
function twice(x) { return add(x, x); }
twice.__closure = () => ({add});
```

Listing 2. An instrumented version of Listing 1, using capture lists. Instrumentation code is highlighted. The ECMAScript 2015 `{add}` shorthand object literal syntax is equivalent to the more explicit `{add: add}` [9].

The first and second points are easily addressed by switching to a serialization format that can represent arbitrary graphs of objects, rather than just trees [7, 8]. YAML is an example of such a format [3].

The third issue is much harder to resolve. In order to serialize a JavaScript function, we need access to the state that constitutes a function object, that is, the function's definition and its list of captured variables. ECMAScript-compliant JavaScript code can access the former, but not the latter. This rules out arbitrary function serialization based on current JavaScript specifications and implementations.

Listing 1 illustrates how ubiquitous variable capture is in JavaScript: function `twice` calls `add` and implicitly captures `add` in the process. To serialize `twice`, a serializer needs to first know that `add` is captured by `twice` and then serialize `add` as well. While `twice` clearly captures `add`, this information is not available to a serializer at run time.

3 FlashFreeze

FlashFreeze's core idea is to equip every variable-capturing function with a list of captured variables—a *capture list.* These capture lists can then be inspected at run time to serialize the functions they belong to.

To ensure that every variable-capturing function has a capture list, FlashFreeze instruments function definitions in the source code it processes. Specifically, FlashFreeze annotates every function definition with an additional nullary function that, when invoked, generates a capture list for the original function. The use of such a *capture list generator* delays capture list creation until necessary for serialization.

Listing 2 shows our approach in action by applying it to our earlier example. In Listing 2, the `twice` function captures `add` and hence FlashFreeze has attached a capture list generator to `twice`'s `__closure` property.

Serializing a FlashFreeze-instrumented function is straightforward: the serializer serializes both the function's definition and its capture list, which the serializer generates on demand. The process of deserializing such a representation is less obvious but equally simple to implement. To deserialize a function object, it suffices to place its definition in a scope in which the names of captured variables are assigned values obtained by decoding the serialized capture list.

3.1 Capture Lists

We define a capture list of a function f to be a dictionary C_f that maps the names of f's captured variables to those variables' values. To construct a capture list C_f, we compute the set difference of the variable names *referenced* by f and the variable names *defined* by f. The capture list maps the resulting set of variables names to their values.

We opt to materialize capture lists lazily: a serializer can create capture lists on demand by invoking any variable-capturing function's capture list generator, injected ahead of time by FlashFreeze. We assign this capture list generator to the function's `__closure` property.

The state of the art in instrumentation techniques for serialization is to populate lexical scope descriptions eagerly [10, 18, 20]. Transposed to capture lists, this would correspond to updating the values in capture lists every time they change. In synthesizing capture list generators, we sidestep the creation and management of scope-like data structures and instead leverage the VM's highly optimized native variable capture implementation to track variables.

3.2 Serialization

To accurately represent arbitrary objects, FlashFreeze encodes serialized objects in a graph data structure, the *serialization graph*. Every object reachable from the value to serialize corresponds to a node in the graph and may refer to other nodes in the graph. A single "root" node identifies the serialized value represented by the graph. All other nodes in the graph are the root's transitive dependencies.

FlashFreeze uses a small number of specialized serialization graph node types to encode JavaScript objects. Different types of objects—from normal objects to regular expressions, arrays and built-in objects—each have dedicated node types.

User-defined functions are no exception. A function node in the serialization graph consists of a function's body, any user-defined properties, and a capture list. The ECMAScript specification provides access to the former two via the reflection functions `Function.prototype.toString()` and `Object.getOwnPropertyNames()` [9]. A capture list can be obtained by evaluating a FlashFreeze-instrumented function's `__closure` property.

Built-in functions such as `Math.sin` do not have a JavaScript implementation, so we cannot serialize them by encoding their function bodies. We overcome this limitation by including an automatically-generated list of built-in functions and objects in the serializer. Whenever an entry in this list is to be serialized, the serializer instead encodes the builtin's name. A deserializer with a compatible list of builtins maps that name to its runtime's version of the builtin.

```
function wrapper(add) {
  return function twice(x){return add(x, x);};
}
```

Listing 3. Generated deserialization code for function `twice` from Listing 2.

3.3 Deserialization

To deserialize a serialization graph, we first create one deserialized object per node in the graph and then populate those objects by deserializing the nodes individually. We finally return the object that corresponds to the graph's root.

In this deserialization scheme, function deserialization is no special case. To deserialize a function, we inject that function's source code into a specially-constructed scope that defines the function's captured variables and maps them to their deserialized values. To create such a scope, FlashFreeze wraps the function in another function that takes the elements of the capture list as arguments. Listing 3 shows what the deserializer's generated code looks like. To instantiate the wrapped function, we deserialize the capture list's elements and call the wrapper function with those elements.

A naive application of the aforementioned approach works well when inter-function dependencies form a directed acyclic graph (DAG). However, it breaks down when two functions capture each other, as is the case with mutually recursive functions. Such functions introduce a chicken and egg problem: when one function captures another function that itself captures the first function to deserialize, we find ourselves in a situation where we cannot deserialize either function due to a cyclic dependency because a call to a wrapper function as in Listing 3 requires a full list of deserialized objects.

To support cyclic dependencies in functions, we associate every deserialized function with a *thunk:* a function object with a body that forwards its arguments to a call to a specially-designated callee property of the thunk. This thunk is created first and its callee property is deserialized later. This allows the thunk to be used as an argument to a wrapper function without requiring the thunk's callee to be deserialized first, solving the chicken and egg problem. To preserve the identity property of objects, all other references to the thunk's callee are also redirected to the thunk.

Once a function's body and captured variables have been deserialized, any user-defined properties are deserialized as well and attached to the thunk object.

3.4 A Note on Mutated Variables

By mapping the names of captured variables to their values in its deserialization algorithm, FlashFreeze implements *by value* capture: every deserialized function gets its own, isolated copy of every captured variable. However, the ECMAScript standard [9] mandates *by reference* capture: functions that share a variable can update it and observe the updated

```
function counter() {
  var c = 0;
  var inc = add => c += add;
  var get = () => c;
  return {inc, get};
}
```

Listing 4. A JavaScript function that creates a counter object.

```
function counter() {
  var c = {value: undefined};
  c.value = 0;
  var inc = add => c.value += add;
  inc.__closure = () => ({c});
  var get = () => c.value;
  get.__closure = () => ({c});
  return {inc, get};
}
```

Listing 5. The same code as in Listing 4, instrumented by FlashFreeze. Added and modified lines are highlighted.

value. This behavior can be observed in Listing 4, which uses by reference capture to create counter objects.

To resolve the conflict between by value and by reference capture, we note that the two variable capture techniques are functionally equivalent for constant variables, that is, variables that are initialized once and don't change after that point. Only mutated captured variables can expose the difference between by value and by reference capture.

FlashFreeze harmonizes its by value capture with JavaScript's by reference capture by applying *assignment conversion* [1] to mutated captured variables prior to the instrumentation pass. Assignment conversion replaces a mutable variable with a constant "cell:" an object that contains a single property representing its value. The contents of such a cell may be mutated, but the cell reference is constant.

To minimize assignment conversion's run time overhead, FlashFreeze statically analyzes its input code and converts only those variables that are both captured and mutated.

Listing 5 shows what the final result looks like in practice when applied to Listing 4: the value stored in the `c` variable is set to a cell object when `c` is defined and all other accesses to `c` are rewritten as accesses to the cell. This transformation makes `c` constant and hence makes the distinction between by value and by reference capture disappear.

3.5 Properties

FlashFreeze's construction offers two desirable, related properties that previous instrumentation-based approaches do not have, because they are based on scope descriptions rather than capture lists. The first property, *precision*, is that a capture list is an exact description of a function's dependencies: no element in a function's capture list is not captured by the function. Thanks to this property, a serializer can effectively serialize a single function and its dependencies without including unnecessary variables from the function's enclosing scopes. This is particularly useful for distributed computing workloads where standalone functions are exchanged.

Another useful property of FlashFreeze is that it preserves the *safe for space* property [25]. That is, FlashFreeze's lazy capture lists do not impose additional restrictions on garbage collection. All objects kept live by capture list generators are already live by virtue of being captured by the generators' associated functions. When a function becomes dead, so does its capture list generator.

```
class Vector2 {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  get length() {
    return Math.sqrt(
      this.x * this.x + this.y * this.y);
  }
}
```

Listing 6. A simple ECMAScript 2015 class definition.

3.6 Limitations

We posit that FlashFreeze's instrumentation construction is semantics-preserving for *self-contained* ECMAScript 5–conforming [9] JavaScript files that do not use `eval` to refer to captured variables; do not use `with` to refer to getter/setter-backed properties; and are agnostic to syntactic source code changes. We will substantiate this claim in section 5 by applying FlashFreeze to the Octane benchmark suite, a large body of ECMAScript 5 code. The self-containedness requirement is due to FlashFreeze's processing of files in isolation; replacing a global variable by a cell in one file does not imply that the same variable is replaced by a cell in another file.

Most features from later standards can be transpiled to ECMAScript 5–conforming JavaScript code [23]. Since FlashFreeze supports ECMAScript 5, it can also handle all transpilable features by extension as long as the transpilation process happens before FlashFreeze's instrumentation code is introduced. Even advanced features such as classes are amenable to instrumentation via transpilation. For instance, consider Listing 6, which defines a class that represents a two-dimensional vector. Transpiling and instrumenting it produces Listing 7. The instrumentation code in Listing 7 correctly identifies the two variables captured by `Vector2`'s class definition: `Object` and `Math`.

ECMAScript 2015 modules [9] cannot be transpiled in the same way and represent a limitation to FlashFreeze's

```
var _a;
var Vector2 = ((_a = function () {
  var _e;
  function Vector2(x, y) {
    this.x = x;
    this.y = y;
  }
  Object.defineProperty(
    Vector2.prototype, "length", {
    get: (_e = function () {
      return Math.sqrt(
        this.x * this.x + this.y * this.y);
    }, _e.__closure = () => ({Math}), _e),
    enumerable: true,
    configurable: true
  });
  return Vector2;
}, _a.__closure = ()=>({Object,Math}),_a)());
```

Listing 7. Transpiled and instrumented class definition. Instrumentation code is highlighted.

instrumentation technique. On the one hand, they allow FlashFreeze to assume that files are self-contained, eliminating the issue of global variable assignment conversion. But on the other hand, they introduce an all but equivalent problem: mutated variables exported by a module. Such mutated variables are equally dangerous to assignment convert due to FlashFreeze's file-by-file approach.

In practice, we have not encountered mutated module-exported variables in the workloads to which we have applied FlashFreeze, but we do consider this to be a serious limitation. In the future, we plan to remove it by rewriting global variables as special properties of their modules.

There are three other inherent limitations to FlashFreeze's approach. First, the `eval` function may inspect or modify a captured variable. Whether an `eval` call will do so is in general not known until said call. FlashFreeze assumes that `eval` never interacts with captured variables, an assumption that a pathological program can violate. We believe this is a mostly theoretical issue as using `eval` is considered bad practice and having `eval` interact with locals doubly so [11].

A second limitation relates to JavaScript's `with` statement. FlashFreeze assumes that all unqualified names refer to variables when synthesizing capture list generators, an assumption that may be violated if `with` is used to introduce a getter/setter-backed property into a scope as an unqualified name. We do not consider this to be a serious issue as using `with` is discouraged and forbidden in strict mode [9, 24].

The third limitation is that FlashFreeze, like any technique rooted in instrumentation, cannot support programs that inspect their own source code. Indeed, FlashFreeze changes said source code.

3.7 Implementation

We implemented FlashFreeze's instrumentation construction as a pair of TypeScript compiler transforms [28], which are available as open-source software.[1] The first transform performs assignment conversion, the second one attaches capture list generators to variable-capturing functions. The TypeScript compiler accepts a superset of JavaScript code, so our implementation applies to both languages [5].

4 Related work

In this section, we will discuss two strands of related work: VM extensions and source-to-source rewriters.

4.1 VM Extensions for Function Serialization

Arguably the most direct way to give JavaScript applications access to captured variables is to extend a JavaScript VM with an API for querying captured variables.

Oh et al. [22] retrofitted Chrome's V8 JavaScript runtime [13] with an API to query captured variables. They then implemented a web application snapshotting tool as a browser extension that relies on their custom API.

Kwon and Moon [19] customized WebKit [2] to include a similar API and implemented snapshotting logic in JavaScript based on their custom API. Their snapshotting tool supports more complex applications than Oh et al.'s work.

Customizing a VM has some key advantages: it opens the door to low static overhead serialization and does not require JavaScript code to be rewritten. The main drawback to nonstandard VM modifications is that they require application developers to have a large degree of control over the software on the user's machine. That is not always the case.

4.2 Source-to-Source Rewriters

Source-to-source rewriters overcome the lack of captured variable information at run time without resorting to VM modifications. They instead rewrite source code to embed additional information so serializers can query functions' captured variables. FlashFreeze is a source-to-source rewriter.

4.2.1 Imagen

Imagen [20] instruments code with statements that at run time generate data structures describing lexical scopes. These lexical scope descriptions correspond closely to variable capture as specified by ECMAScript [9]. FlashFreeze's capture lists, on the other hand, are similar to how variable capture works in the C++ programming language specification [15].

Listing 8 shows what Imagen's construction looks like when applied to our example from Listing 1. The `scope` object describes the lexical scope in which `add` and `twice` are defined. When instructed to serialize a function, Imagen queries the function's `parentScope` property to generate

[1]See https://github.com/nokia/ts-serialize-closures.

```
var scope = new Object();
function add(x, y) { return x + y; }
scope.add = add;
add.parentScope = scope;
function twice(x) { return add(x, x); }
twice.parentScope = scope;
```

Listing 8. IMAGEN's instrumentation construction, applied to Listing 1. Instrumentation code is highlighted.

```
function counter() {
  var scope = new Object();
  var c = 0;
  scope.c = c;
  var inc = add => {
    c += add;
    scope.c = c;
    return c;
  };
  inc.parentScope = scope;
  var get = () => c;
  get.parentScope = scope;
  return {inc, get};
}
```

Listing 9. IMAGEN's instrumentation construction, applied to Listing 8. Instrumentation code is highlighted.

code that reconstructs the function to restore as well as its enclosing scopes. Running the code deserializes the function.

IMAGEN's approach is fundamentally eager. Lexical scope descriptions are created and populated regardless of whether they are actually used. Listing 9 exemplifies this behavior: every time the c variable is updated, `scope.c` is also updated. Due of this eagerness, the overhead associated with lexical scope descriptions applies as much to function objects that are never serialized as to the ones that are serialized.

4.2.2 ThingsMigrate

ThingsMigrate [10] is a recent project that also implements function serialization by instrumenting source code files.

ThingsMigrate's instrumentation construction was initially roughly the same as IMAGEN's [10, 20]. The latest version of ThingsMigrate [17] uses a slightly different construction that no longer eagerly updates captured variables when they change. Lexical scope descriptions are still created eagerly by the new construction. Listing 10 applies ThingsMigrate's new construction to our running example.

ThingsMigrate's construction calls into a JavaScript runtime library to construct scope descriptions. Source code instrumented by ThingsMigrate depends on that library; instrumented files will not execute without it. IMAGEN and FlashFreeze do not introduce such dependencies.

```
Σ.setExtractor(() => this.capture({}, {}))
  .hoist(add, Σ).hoist(twice, Σ);
function add(x, y) {
  var Σ_add = new Σ.Scope(
    this, Σ, add, () =>
      this.capture({x, y}, {}));
  return x + y;
}
function twice(x) {
  var Σ_twice = new Σ.Scope(
    this, Σ, twice, () =>
      this.capture({x}, {}));
  return add(x, x);
}
```

Listing 10. ThingsMigrate's new instrumentation construction, applied to Listing 1. Instrumentation code is highlighted.

```
var $fsm1 = $fsm.create();
$fsm1.add = function add(x, y) {return x+y;}
$fsm1.twice = function twice(x) {
  return $fsm1.add(x, x);
}
```

Listing 11. FSM's instrumentation construction, applied to Listing 1. This example assumes that `add` and `twice` are not global variables.

ThingsMigrate's authors discovered that IMAGEN's serialization algorithm does not support nested variable-capturing functions [10]. To correctly serialize such functions, ThingsMigrate implements a more complex serialization algorithm that reasons about *trees* of scopes; IMAGEN only considers scope *chains*, that is, paths in scope trees.

4.2.3 Fast Snapshot Migration

FSM [18] is a low-overhead instrumentation construction for creating lexical scope descriptions. Whereas IMAGEN's construction maintains mirror versions of lexical scopes, FSM replaces each lexical scope with an object and rewrites all variable accesses to use that object. Listing 11 applies FSM to our running example.

A notable exception is the global scope: FSM accesses globals by cleverly inspecting the JavaScript VM's global object. Hence, FSM need not instrument functions that refer to globals only. As we will show in the evaluation section, this approach pays dividends for FSM on the Octane benchmark suite. Indeed, only two FSM-instrumented benchmarks—Box2D and RegExp—show significant Octane score reductions compared to the uninstrumented benchmark.

Table 1. A comparison of JavaScript object graph serialization techniques; static overhead is measured by Octane scores.

Technique	JSON	YAML	Oh et al. [22]	Kwon and Moon [19]	IMAGEN [20]	ThingsMigrate [10]	FSM [18]	FlashFreeze
Graphs	✗	✓	✓	✓	✓	✓	✓	✓
Functions	✗	✗	✓	✓	✓	✓	✓	✓
Nested functions	✗	✗	✗	✓	✗	✓	✓	✓
Static overhead	None	None	None	None	High	High	Low	Low
No VM changes	✓	✓	✗	✗	✓	✓	✓	✓
No memory leaks	✓	✓	✓	✓	✓	✓	✗	✓
Precise dependencies	N/A	N/A	✗	✗	✗	✗	✗	✓
Safe for space	✓	✓	✓	✓	✗	✗	✗	✓
No runtime library	✓	✓	✓	✓	✓	✗	✗	✓
No scope tree	✓	✓	✓	✗	✓	✗	✗	✓

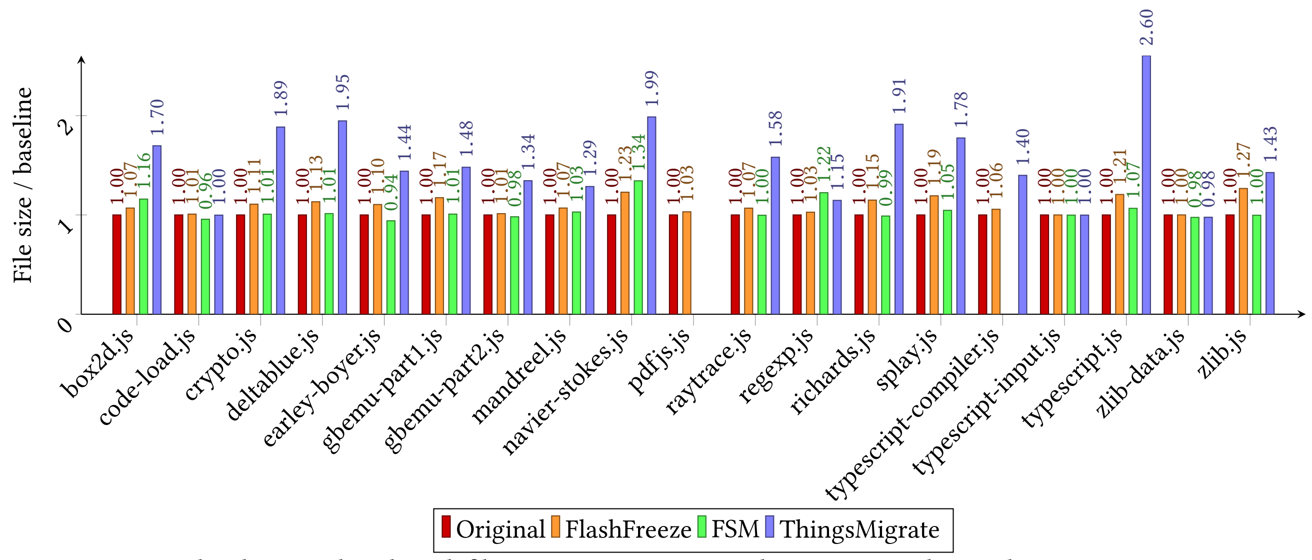

Figure 1. Normalized Octane benchmark file sizes. Lower scores are better. Missing bars indicate an error or omission.

FSM is a work in progress. Its construction currently causes memory leaks: scope descriptions are stored in a table, keeping the descriptions alive even after the scopes themselves become dead. Consequently, scope descriptions and any objects they refer to cannot be garbage-collected.

As with ThingsMigrate, FSM-instrumented code calls into a separate JavaScript runtime library to construct lexical scope descriptions. This introduces a run-time dependency on that library.

4.3 Comparison

Table 1 summarizes the characteristics of the various approaches discussed in this paper, including FlashFreeze's.

We classify ThingsMigrate as high-overhead based on the measurements we will describe in section 5. We also classify IMAGEN as a high-overhead technique as its construction is an earlier, less optimized version of ThingsMigrate's [10, 17, 20].

The "precise dependencies" and "safe for space" rows refer to the eponymous properties from subsection 3.5. A cross mark in the "no runtime library" row specifies if instrumenting a file makes it dependent on a separate runtime library. A check mark means the opposite. "No scope tree" means that the tool's serialization and deserialization algorithms give no special consideration to trees of scopes.

5 Evaluation

We now use the Octane JavaScript benchmark suite [12] to evaluate FlashFreeze's instrumentation construction. We split each benchmark into four versions: an unmodified version, an FSM-instrumented version, a ThingsMigrate-instrumented version and a FlashFreeze-instrumented version. We then measure three quantities for each benchmark variant: (1) whether the benchmark throws an error, (2) source code file sizes and (3) the benchmark's Octane score.

Benchmark errors allow us to assess an instrumentation technique's feature-completeness. File sizes serve as a coarse measure of source code storage and transmission costs. Octane scores measure throughput and latency.

Benchmarks were run using NodeJS v10.15.2 installed via the `nodejs` APT package [26] on an Ubuntu 19.04 machine sporting an Intel® Core™ i7-6700K CPU. We used ThingsMigrate's latest version: v2.0.0. FSM-instrumented Octane benchmarks were provided to us by FSM's author.

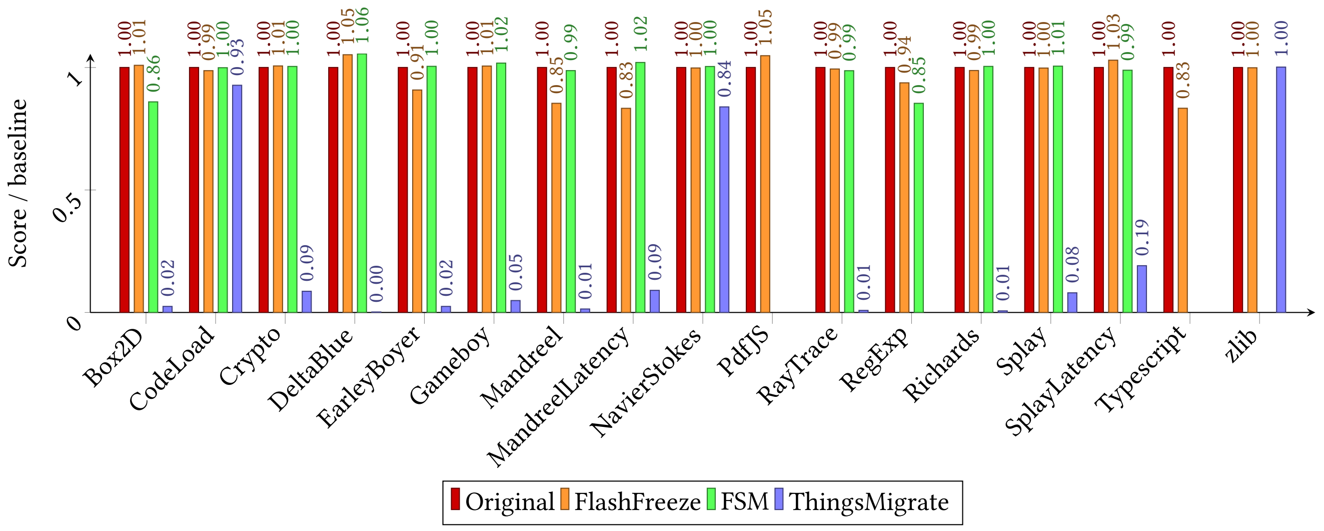

Figure 2. Normalized Octane benchmark scores. Higher scores are better. Missing bars indicate an error or omission.

5.1 Feature-Completeness

Octane represents a 12.71 MiB body of JavaScript programs, ranging from the asm.js [14] Mandreel benchmark to a PDF reader [21] and a TypeScript compiler. These benchmarks have complex behavior. For instance, the TypeScript benchmark makes the eponymous compiler compile itself [12].

FlashFreeze correctly instruments all Octane benchmarks. ThingsMigrate's instrumentation breaks the semantics of RegExp and TypeScript; it also throws an error when made to instrument PdfJS. The FSM-instrumented benchmarks we were provided do not include PdfJS, TypeScript and zlib.

5.2 Benchmark File Sizes

Figure 1 shows the static overhead of instrumentation in terms of file size relative to the uninstrumented benchmarks' sizes. FlashFreeze's overhead is always modest: benchmark file size growth stays in the 0% to 27% range. FSM's file size growths are in a similar range from 0% to 33%. ThingsMigrate's file size growth peaks at 160%. FlashFreeze, FSM and ThingsMigrate's mean file size growths are 10%, 4% and 55%.

5.3 Octane Scores

Figure 2 summarizes Octane scores assigned to each benchmark version. The numbers on top of the bars indicate the Octane score assigned to a benchmark, normalized to the uninstrumented equivalent's score. Both FlashFreeze and FSM outperform ThingsMigrate in terms of Octane scores.

An Octane benchmark instrumented by ThingsMigrate loses on average 76% of its Octane score. FlashFreeze's instrumentation logic results in a mean score loss of only 3%. FSM has an even lower loss of 1%.

On some benchmarks, the instrumented benchmarks attain higher scores than their uninstrumented counterparts. This effect is due to measurement noise. FlashFreeze's poor performance on Mandreel stems from Mandreel being a low-level `asm.js` program whose often-accessed global `heap` variable is assignment converted. FSM performs well on Mandreel because the benchmark overwhelmingly consists of functions that capture only globals—such functions are not modified by FSM. Box2D and RegExp contain mostly functions that capture locals, which FSM does instrument. FlashFreeze incurs lower overheads on these benchmarks.

6 Conclusion

We introduced FlashFreeze, a lightweight instrumentation construction based on lazy capture lists. FlashFreeze is a general-purpose tool that injects no runtime dependencies into instrumented files. This property allowed us to easily use FlashFreeze for a distributed computing application.

FlashFreeze's capture lists replace the complicated notion of lexical scopes and shared captured variables in other (de)serializers, allowing for simple (de)serialization logic. Additionally, capture lists preserve the safe for space property and provide a precise description of a function's dependencies, allowing serializers to serialize a single function without serializing all global program state in the process.

Experimental results show that FlashFreeze achieves a mean file size growth of 10% and a mean Octane benchmark score reduction of 3%. Those metrics are 55% and 76% for the state-of-the-art ThingsMigrate tool; 4% and 1% for the work-in-progress FSM technique. Of these three techniques, only FlashFreeze has been shown to instrument the entire Octane benchmark suite in a semantics-preserving way.

References

[1] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN '86)*. 219–233.

[2] Apple Inc. 2019. WebKit: a fast, open source web browser engine. https://webkit.org/ Accessed on March 28, 2019.

[3] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. YAML Ain't Markup Language (YAML™) version 1.2. https://yaml.org/spec/cvs/spec.pdf Accessed on May 19, 2019.

[4] John K. Bennett. 1987. The Design and Implementation of Distributed Smalltalk. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. 318–330.

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming*. Springer, 257–281.

[6] Akmal B Chaudhri. 2017. Apache Ignite Tip: Peer Class Loading Deployment Magic. https://www.gridgain.com/resources/blog/apacher-ignitetm-tip-peer-class-loading-deployment-magic Accessed on May 18, 2019.

[7] Douglas Crockford. 2018. JSON in JavaScript. https://github.com/douglascrockford/JSON-js Accessed on May 13, 2019.

[8] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. 2014. Fuel: A fast general purpose object graph serializer. *Software: Practice and Experience* 44, 4 (2014), 433–453.

[9] ECMA International. 2015. *ECMAScript 2015 Language Specification*. https://www.ecma-international.org/ecma-262/6.0/index.html Accessed on May 17, 2019.

[10] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. 2018. ThingsMigrate: Platform-independent migration of stateful JavaScript IoT applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 94–105.

[12] Google Developers. 2017. Octane: the JavaScript benchmark suite for the modern web. https://developers.google.com/octane/ Accessed on March 28, 2019.

[13] Google Developers. 2019. V8 JavaScript engine. https://v8.dev/ Accessed on March 28, 2019.

[14] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js: Working Draft. http://asmjs.org/spec/latest/ Accessed on May 17, 2019.

[15] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). International Organization for Standardization, Geneva, Switzerland. 1605 pages. https://www.iso.org/standard/68564.html

[16] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.

[17] Kumseok Jung, Julien Gascon-Samson, Aarti Kashyap, Xuejie Tang, Karthik Pattabiraman, and marusshi. 2019. ThingsJS. https://github.com/DependableSystemsLab/ThingsJS Accessed on May 20, 2019.

[18] Jae-Yun Kim, Hyeon-Jae Lee, and Soo-Mook Moon. 2018. Work-in-Progress: Fast Snapshot Migration Using Static Code Instrumentation. In *2018 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–2.

[19] Jin-woo Kwon and Soo-Mook Moon. 2017. Web application migration with closure reconstruction. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 133–142.

[20] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. 2013. Imagen: runtime migration of browser sessions for JavaScript web applications. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 815–826.

[21] Mozilla and individual contributors. 2016. PDF.js: A general-purpose, web standards-based platform for parsing and rendering PDFs. https://mozilla.github.io/pdf.js/ Accessed on May 17, 2019.

[22] JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. 2015. Migration of web applications with seamless execution. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 173–185.

[23] Narayan Prusty. 2015. *Learning ECMAScript 6*. Packt Publishing Ltd.

[24] Axel Rauschmayer. 2011. JavaScript's `with` statement and why it's deprecated. https://2ality.com/2011/06/with-statement.html Accessed on August 30, 2019.

[25] Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-space Closure Conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (Jan. 2000), 129–161.

[26] Ubuntu Developers. 2019. Package `nodejs` 10.15.2. https://packages.ubuntu.com/disco/nodejs Accessed on August 31, 2019.

[27] Wolfgang Van Raemdonck, Tom Van Cutsem, Kyumars Sheykh Esmaili, Mauricio Cortes, Philippe Dobbelaere, Lode Hoste, Eline Philips, Marc Roelands, and Lieven Trappeniers. 2017. Building connected car applications on top of the world-wide streams platform. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*. ACM, 315–318.

[28] Kris Zyp. 2017. How to write a TypeScript transform (plugin). https://dev.doctorevidence.com/how-to-write-a-typescript-transform-plugin-fc5308fdd943 Accessed on March 29, 2019.