JITProf: Pinpointing JIT-Unfriendly JavaScript Code

Liang Gong1, Michael Pradel2, and Koushik Sen1
1 EECS Department, University of California, Berkeley, USA
2 Department of Computer Science, TU Darmstadt, Germany,
1 (gongliang13, ksen)@cs.berkeley.edu,
2 michael@binaervarianz.de

ABSTRACT

Most modern JavaScript engines use just-in-time (JIT) compilation to translate parts of JavaScript code into efficient machine code at runtime. Despite the overall success of JIT compilers, programmers may still write code that uses the dynamic features of JavaScript in a way that prohibits profitable optimizations. Unfortunately, there currently is no way to measure how prevalent such JIT-unfriendly code is and to help developers detect such code locations. This paper presents JITProf, a profiling framework to dynamically identify code locations that prohibit profitable JIT optimizations. The key idea is to associate meta-information with JavaScript objects and code locations, to update this information whenever particular runtime events occur, and to use the meta-information to identify JIT-unfriendly operations. We use JITProf to analyze widely used JavaScript web applications and show that JIT-unfriendly code is prevalent in practice. Furthermore, we show how to use the approach as a profiling technique that finds optimization opportunities in a program. Applying the profiler to popular benchmark programs shows that refactoring these programs to avoid performance problems identified by JITProf leads to statistically significant performance improvements of up to 26.3% in 15 benchmarks.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Testing tools, Debugging aids, Monitors, Tracing

Keywords

JavaScript, profiler, JITProf, dynamic analysis, just-in-time compilation, Jalangi

1. INTRODUCTION

JavaScript is the most widely used client-side language for writing web applications. It is also getting increasingly popular on mobile and desktop platforms. To further improve performance, modern JavaScript engines use just-in-time (JIT) compilation [13, 18, 8, 4], which translates and optimizes JavaScript code into efficient machine code while the program executes.

Despite the overall success of JIT compilers, programmers may write code using JavaScript dynamic features in a way that prohibits profitable JIT optimizations. We call such JavaScript code JIT-unfriendly code. Previous research [39] shows that programmers extensively use those dynamic features, including dynamic addition and deletion of object properties. However, an important premise for effective JIT optimization is that programmers use the dynamic features of JavaScript in a regular and systematic way. For code that satisfies this premise, the JavaScript engine generates and executes efficient machine code. Otherwise, the engine must fall back to slower code or to interpreting the program, which can lead to significant performance penalties, as noticed by developers [2, 3, 1].

Even though there is evidence that JIT-unfriendly code exists, there currently is no way to identify JIT-unfriendly code locations and to measure how prevalent the problem is. Addressing these challenges helps improving the performance of JavaScript programs in two ways. First, a technique to identify JIT-unfriendly code locations in a program helps application developers to avoid the problem. Specialized profilers for other languages and performance problems [28, 47, 32] show that pinpointing developers to optimization opportunities is valuable. Second, empirical evidence on the prevalence of JIT-unfriendly code helps developers of JavaScript engines to focus their efforts on the most important patterns of JIT unfriendliness. Recent work shows that small modifications in the JavaScript engine can have a dramatic impact on performance [4].

This paper addresses the challenge of identifying and measuring JIT-unfriendliness through a dynamic analysis, called JITProf, that identifies code locations that prohibit profitable JIT optimizations. The key idea is to identify potentially JIT-unfriendly operations by analyzing runtime execution patterns and to report code locations that could potentially cause slowdown. JITProf associates meta-information with JavaScript objects and code locations, updates this information whenever particular runtime events occur, and uses the meta-information to identify JIT-unfriendly operations. For example, JITProf tracks hidden classes and inline cache misses, which are two important concepts in JIT optimization, by associating a hidden class with every
JavaScript object and a cache-miss counter with every code location that accesses an object property.

A key advantage of our approach is that it does not hard-code a set of checks for JIT unfriendliness into a particular JavaScript engine but instead provides an extensible, engine-independent framework for checking various JIT-unfriendly code patterns. We implement JITProf as an open-source prototype framework that instruments JavaScript code by source-to-source transformation, so that the instrumented code identifies JIT-unfriendly code locations at runtime. We instantiate the JITProf framework for seven JIT-unfriendly code patterns that cause performance problems in the Firefox and Chrome browsers. Supporting additional patterns does not require detailed knowledge of the internals of a JIT compiler. Instead, understanding the JIT-unfriendly code pattern at a high-level is sufficient to write JavaScript code that use JITProf’s API. This user-level extensibility is important because JIT compilers evolve rapidly and different JIT compilers employ different optimizations.

We apply JITProf in two ways. First, we conduct an empirical study involving popular websites and benchmarks to understand the prevalence of JIT-unfriendly code patterns in practice. We find that JIT unfriendliness is common in both websites and benchmarks, and we show the relative prevalence of different JIT-unfriendly code patterns. Our results suggest that work on addressing these code patterns by modifying the applications is worthwhile.

Second, we apply the JITProf approach as a profiling technique to find optimization opportunities in a program and evaluate whether using these opportunities improves the program’s performance. We show that JITProf effectively detects various JIT-unfriendly code locations in the SunSpider and Octane benchmarks, and that refactoring these locations into JIT-friendly code yields statistically significant improvements of execution time in 15 programs. The improvements, which range between 1.1% and 26.3%, exist in Firefox and Chrome, both of which are tuned towards the JavaScript engine but instead provides an extensible, engine-independent framework that analyzes runtime information with locations. JITProf enables profilers to attach arbitrary objects to objects of the program under analysis) with objects. JITProf is designed to detect occurrences of these patterns. JITProf’s design is motivated by four recurring patterns described in this paper.

• We use JITProf to conduct an empirical study of JIT-unfriendly code, showing that it is prevalent both in websites and benchmarks.
• We use JITProf as a profiler that pinpoints JIT-related optimization opportunities to the developer, and show that the approach finds valuable optimization opportunities in 15 out of 39 JavaScript benchmark programs.
• We make our implementation available as open-source (BSD license) to provide a platform for future research: https://github.com/Berkeley-Correctness-Group/JITProf

2. APPROACH

This section describes JIT-unfriendly patterns known to exist in state-of-the-art JavaScript engines and presents our approach to detect occurrences of these patterns.

2.1 Framework Overview

We design JITProf as an extensible framework that provides a reusable API that accommodates not only today’s but also future JIT unfriendly code patterns. A generic approach is crucial because JavaScript engines are a fast-moving target. The API, summarized in Table 1, defines functions that profilers can implement and that are called by the framework, as well as functions that the profilers can call. JITProf’s design is motivated by four recurring properties of JIT-unfriendly code patterns from which we derive requirements for profilers to detect them.

Runtime Events: All patterns are related to particular runtime events and profilers need a way to keep track of these events. JITProf supports a set of runtime events for which profilers can register. At every occurrence of a runtime event, the framework calls into the profiler and the profiler can handle the event. The upper part of Table 1 lists the runtime events that profilers can register for. For example, a profiler can implement the getProp() function, which gets called on every property read operation during the execution. Our implementation supports more runtime events than the events listed in Table 1; for brevity, we focus on events needed for the seven JIT-unfriendly patterns described in this paper.

Associate Shadow Information: Some patterns are related to particular runtime objects and profilers need a way to associate shadow information (meta-information invisible to the program under analysis) with objects. JITProf enables profilers to attach arbitrary objects to objects of the program under test. v.meta allows profilers to access the shadow-information associated with a particular runtime value v. Moreover, patterns are related to particular code locations and profilers need a way to associate shadow-information with locations. JITProf enables profilers to attach arbitrary information to code locations through the l.meta function, which returns the shadow-information associated with a location l. In addition, JITProf provides profilers to keep track of how often a code location is involved in a JIT-unfriendly operation, which we find to be an effective way to identify JIT-unfriendly locations. Therefore, JITProf associates a zero-initialized counter with locations that may execute a JIT-unfriendly operation. We call this counter the unfriendliness counter. Whenever a profiler observes a JIT-unfriendly operation, it increments the unfriendliness counter of the operation via the incrCnt() function.

Prioritize JIT-unfriendly Code: Profilers need a way to prioritize potentially JIT-unfriendly code locations, e.g., to help developers focus on the most promising optimization opportunities. JITProf provides a default ranking strategy that reports locations sorted by their unfriendliness counter (in descending order). Unless otherwise mentioned, the profilers described in this paper use the default ranking strategy.

Sampling: Profiling for JIT-unfriendly code locations in a naive way can easily cause very high overhead, even for programs that do not suffer from JIT-unfriendliness. To reduce the overhead, JITProf uses a sampling strategy that adapts the profiling effort to the amount of JIT-unfriendliness observed at a particular location (Section 2.3).
A common source of JIT-unfriendly behavior are code locations that apply an operation to different types at different executions of the location. We call such operations *polymorphic operations*.

**Micro-benchmark** We illustrate each JIT-unfriendly code pattern with a simple example. The plus operation at line 1 of Figure 1 operates on both numbers and strings. The performance of the example can be significantly improved by splitting the function that operates on numbers and a function that operates on strings, as shown on the right of Figure 1. The modified code runs 92.1% and 72.2% faster in Firefox and Chrome, respectively.

**Explanation** This change enables the JavaScript engine to execute specialized code for the plus operation because the change turns a polymorphic operation into two monomorphic operations, i.e., operations that always execute on the same types of operands. For example, the JIT compiler can optimize the monomorphic plus at line 1 of the modified example into a few quick integer instructions and inline these instructions at the call site of f. In contrast, the JIT compiler does not optimize the original code because the types of operands change every time line 1 executes.

**Profiling** To detect performance problems caused by polymorphic operations, Profiler PO in Table 2 tracks the types of operands involved in unary and binary operations. The profiler maintains for each code location that performs a unary or binary operation the most recently observed type(s) lastType1 and lastType2 of the left (and right) operand.

Whenever the program performs a binary operation, the profiler checks whether the types of the operands match lastType1 and lastType2. If at least one of the current types differs from the respective stored type, then the profiler increments the unfriendliness counter, and it updates lastType1 and lastType2 with the current types. The profiler performs similar checks for unary operations.

To rank locations for reporting, the profiler combines the framework’s default ranking with an estimate of how profitable it is to fix a problem. For this estimate, the profiler maintains for each location a histogram of observed types. The histogram maps a type or pair of types to the number of times that this type has been observed. The profiler reports all code locations with a non-zero unfriendliness counter, ranked by the sum of the counter and the number (we call it C2) of occurrences of the second most frequently observed type at the location. This approach is a heuristic to break ties when multiple locations have similar numbers of JIT-unfriendly operations. The rationale is that the location with a larger C2 is likely to be more profitable to fix because making the two most frequent types consistent can potentially avoid more JIT-unfriendliness.

For the example in Figure 1, the profiler warns about the polymorphic operation at line 1 because the types of its operands always differ from the previously observed types.
(PO) Polymorphic Operations:

unary(l, *, v, *)  if (type(v) ≠ l.meta.lastType) then incrCtr(l)
    l.meta.lastType ← type(v)
    l.meta.histo.add(type(v))

binary(l, *, v1, v2, *)  if (type(v1) ≠ l.meta.lastType1 ∨ type(v2) ≠ l.meta.lastType2) then incrCtr(l)
    l.meta.lastType1 ← type(v1)
    l.meta.lastType2 ← type(v2)
    l.meta.histo.add(type(v1), type(v2))

(BOU) Binary Operations on undefined:

binary(l, *, v1, v2, *)  if v1 = undefined ∨ v2 = undefined then incrCtr(l)

(NCA) Non-contiguous Arrays:

putProp(l, base, prop, v)  if isArray(base) ∧ isNumber(prop) ∧ (prop < 0 ∨ prop > base.length) then incrCtr(l)

(UAE) Accessing Undefined Array Elements:

getProp(l, base, prop, v)  if (isArray(base) ∧ isNumber(prop) ∧ prop ≠ base) then incrCtr(l)

(NNA) Storing Non-numeric Values in Numeric Arrays:

newObject(l, v)  if isArray(v) then containsNonNumeric(v) then v.meta.state ← NON
    else if allNumeric(v) then v.meta.state ← NUM
    else v.meta.state ← UNK
    oldState ← l.meta
    updateState(base.meta, v)
    if oldState = NUM ∧ base.meta.state = NON then incrCtr(l)

(IOL) Inconsistent Object Layouts (“HC” means hidden class):

newObject(l, v)  v.meta ← getOrCreateHC(v)
    putProp(l, base, p, v)  base.meta ← getOrCreateHC(v)
        if base.meta ≠ l.meta.cachedHC ∨ p ≠ l.meta.cachedProp then incrCtr(l)
        l.meta.cachedProp ← base.meta
        l.meta.cachedHC ← base.meta
        l.meta.histo.add(base.meta)
    getProp(l, base, p, v)  if base.meta ≠ l.meta.cachedHC ∨ p ≠ l.meta.cachedProp then incrCtr(l)
        l.meta.cachedHC ← base.meta
        l.meta.cachedProp ← p
        l.meta.histo.add(base.meta)

(GA) Unnecessary Use of Generic Arrays:

newObject(l, v)  if isArray(v) then v.meta ← initArrayMetaInfo()
    putProp(*, base, prop, v)  if isArray(base) then updateTypeFlags(base.meta, prop, v)
    unary(*, op, v, *)  if isArray(v) ∧ op = "typeof" then setFlag(v.meta, "typeof")
    invokeFun(*, f, base, *, *)  if isArray(base) then setBuiltinsUsed(v.meta, f)

Figure 3: Example of non-contiguous arrays (left) and improved code (right).

For the example in Figure 2, the profiler warns about line 3 because the first operand of the operation is frequently observed to be undefined.

2.2.3 Non-contiguous Arrays

In JavaScript, arrays can have “holes”, i.e., the elements at some indexes between zero and the end of the array may be uninitialized. Such non-contiguous arrays cause slowdown.

Micro-benchmark The code on the left of Figure 3 initializes an array in reverse order so that every write at line 4 is accessing a non-contiguous array. Modifying this code so that the array grows contiguously leads to an improvement of 97.5% and 90.2% in Firefox and Chrome, respectively.

Explanation Non-contiguous arrays are JIT-unfriendly for three reasons. First, JavaScript engines use a slower implementation for non-contiguous arrays than for contiguous arrays. Dense arrays, where all or most keys are contiguous starting from zero, use linear storage. Sparse arrays, where keys are non-contiguous, are implemented as hash tables, and looking up elements is relatively slow. Second, the JavaScript engine may change the representation of an array if its density changes during the execution. Third, JIT compilers speculatively specialize code under the assumption that arrays do not have holes and fall back on slower code if this assumption fails [19].

Profiling To detect performance problems caused by non-contiguous arrays, Profiler NCA in Table 2 tracks for each property-setting code location how often a code location causes slowdown.

Table 2: Profilers to find JIT-unfriendly code locations.

<table>
<thead>
<tr>
<th>Runtime event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PO) Polymorphic Operations:</td>
<td></td>
</tr>
<tr>
<td>unary(l, *, v, *)</td>
<td>if (type(v) ≠ l.meta.lastType) then incrCtr(l)</td>
</tr>
<tr>
<td>binary(l, *, v1, v2, *)</td>
<td>if (type(v1) ≠ l.meta.lastType1 ∨ type(v2) ≠ l.meta.lastType2) then incrCtr(l)</td>
</tr>
<tr>
<td>(BOU) Binary Operations on undefined:</td>
<td></td>
</tr>
<tr>
<td>binary(l, *, v1, v2, *)</td>
<td>if v1 = undefined ∨ v2 = undefined then incrCtr(l)</td>
</tr>
<tr>
<td>(NCA) Non-contiguous Arrays:</td>
<td></td>
</tr>
<tr>
<td>putProp(l, base, prop, v)</td>
<td>if isArray(base) ∧ isNumber(prop) ∧ (prop &lt; 0 ∨ prop &gt; base.length) then incrCtr(l)</td>
</tr>
<tr>
<td>(UAE) Accessing Undefined Array Elements:</td>
<td></td>
</tr>
<tr>
<td>getProp(l, base, prop, v)</td>
<td>if (isArray(base) ∧ isNumber(prop) ∧ prop ≠ base) then incrCtr(l)</td>
</tr>
<tr>
<td>(NNA) Storing Non-numeric Values in Numeric Arrays:</td>
<td></td>
</tr>
<tr>
<td>newObject(l, v)</td>
<td>if isArray(v) then containsNonNumeric(v) then v.meta.state ← NON</td>
</tr>
<tr>
<td>putProp(l, base, prop, v)</td>
<td>if allNumeric(v) then v.meta.state ← NUM</td>
</tr>
<tr>
<td>newObject(l, v)</td>
<td>else v.meta.state ← UNK</td>
</tr>
<tr>
<td>oldState ← l.meta</td>
<td></td>
</tr>
<tr>
<td>updateState(base.meta, v)</td>
<td></td>
</tr>
<tr>
<td>if oldState = NUM ∧ base.meta.state = NON then incrCtr(l)</td>
<td></td>
</tr>
<tr>
<td>(IOL) Inconsistent Object Layouts (“HC” means hidden class):</td>
<td></td>
</tr>
<tr>
<td>newObject(l, v)</td>
<td>v.meta ← getOrCreateHC(v)</td>
</tr>
<tr>
<td>putProp(l, base, p, v)</td>
<td>base.meta ← getOrCreateHC(v)</td>
</tr>
<tr>
<td>getProp(l, base, p, v)</td>
<td>if base.meta ≠ l.meta.cachedHC ∨ p ≠ l.meta.cachedProp then incrCtr(l)</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.cachedProp ← base.meta</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.cachedHC ← base.meta</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.histo.add(base.meta)</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>if base.meta ≠ l.meta.cachedHC ∨ p ≠ l.meta.cachedProp then incrCtr(l)</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.cachedHC ← base.meta</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.cachedProp ← p</td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>l.meta.histo.add(base.meta)</td>
</tr>
<tr>
<td>(GA) Unnecessary Use of Generic Arrays:</td>
<td></td>
</tr>
<tr>
<td>newArray(l, v)</td>
<td>if isArray(v) then v.meta ← initArrayMetaInfo()</td>
</tr>
<tr>
<td>putProp(*, base, prop, v)</td>
<td>if isArray(base) then updateTypeFlags(base.meta, prop, v)</td>
</tr>
<tr>
<td>unary(*, op, v, *)</td>
<td>if isArray(v) ∧ op = &quot;typeof&quot; then setFlag(v.meta, &quot;typeof&quot;)</td>
</tr>
<tr>
<td>invokeFun(*, f, base, *, *)</td>
<td>if isArray(base) then setBuiltinsUsed(v.meta, f)</td>
</tr>
</tbody>
</table>

```javascript
1 for (var j=0; j<400; j++) {
2    var array = [ ];
3 for (var i=5000; i>0; i--){
4    array[i] = i;
5    }
6 }
```
1. var array = [], sum = 0;
2. for(var i=0;i<100;i++) {
3.     array[i] = 1;
4.     array[i] = 1/10;
5.     array[i] = i/10;
6. }
7. var len = array.length;
8. while (array[i]) {
9.     sum += array[i];
10.     i++;
11. }

---

2.2.4 Accessing Undefined Array Elements

Another array-related source of inefficiency is accessing an uninitialized, deleted, or out of bounds array element.

**Micro-benchmark** The code in Figure 4 creates an array and repeatedly iterates through it. The original code on the left checks whether it has reached the end of the array by checking whether the current element is defined, i.e., the code accesses an uninitialized array element each time it reaches the end of the while loop. The modified code on the right avoids accessing an undefined array, which improves performance by 73.9% and 70.2% in Firefox and Chrome, respectively.

**Explanation** Similar to Section 2.2.3.

**Profiling** To find performance problems caused by accessing undefined array elements, Profiler UAE in Table 2 tracks all operations that read array elements. The unfriendliness counter represents how often a code location reads an undefined array element. The profiler checks for each get property operation that reads an array element from base whether the property *prop* is an index if the array. If the check fails, the program accesses an undefined array element, and the profiler increments the unfriendliness counter.

For the example in Figure 4, the profiler warns about line 7 because it reads an undefined array element every time the while loop terminates.

---

2.2.5 Storing Non-numeric Values in Numeric Arrays

JavaScript arrays may contain elements of different types. For good performance, programmers should avoid storing non-numeric values into an otherwise numeric array.

**Micro-benchmark** The code on the left of Figure 5 creates a large array of numeric values and then stores a non-numeric value into it. The modified code avoids storing a non-numeric value, which improves performance by 14.9% and 83.8% in Firefox and Chrome, respectively.

**Explanation** If a dense array contains only numeric values, such as 31-bit signed integers or doubles, then the JavaScript engine can represent the array efficiently as a fixed sized C-like array of integers or doubles, respectively. Changing the representation of the array from a fixed-sized integer/double array to an array of non-numeric values is an expensive operation.

**Profiling** To detect performance problems caused by transforming numeric arrays into non-numeric arrays, Profiler NNA in Table 2 maintains for each array a finite state machine (Figure 7). When an array gets created, the profiler uses the *newObject* function to store the initial state of the array as the array’s shadow-information. The state is initialized to *unknown* if the array is empty or if all elements are uninitialized. If all the elements of the array are numeric, then the state is initialized to *numeric*. Otherwise, the state is initialized to *non-numeric*. The profiler updates the state of an array whenever the program writes into the array through a put property operation, as shown in Figure 7. The profiler increments the unfriendliness counter of a code location that writes an array element when transitioning from *numeric* to *non-numeric*.

For the example in Figure 5, the profiler warns about line 4 because a numeric array gets a non-numeric value.

---

2.2.6 Inconsistent Object Layouts

A common pattern of JIT-unfriendly code is to construct objects of the same type in a way that forces the compiler to use multiple representations for this type. Such *inconsistent object layouts* prevent an optimization that specializes property accesses for recurring object layouts.

**Micro-benchmark** The program in Figure 6 has a constructor function *C* that creates objects with two properties *a* and *b*. Depending on the value of *i*, these properties are created in different orders. The main loop of the program repeatedly creates *C* instances and passes them to *sum*, which accesses the two properties of the object. The expression `base[p1]` returns the value of the property whose name is stored as a string in the variable *p1*. The performance of the example can be significantly improved by swapping lines 6 and 7. The modified code, given on the right of

---

<Figure 4: Accessing undefined array elements.>

```javascript
var array = [];
for(var i=0;i<10000000;i++) {
    array[i] = i/10;
    array[i] = 1;
    array[i] = 1/10;
}
var len = array.length;
while (array[i]) {
    sum += array[i];
    i++;
}
```

<Figure 5: Example of storing non-numeric values into numeric arrays.>

```javascript
var array = [];
for(var i=0;i<100;i++) {
    array[i] = 1;
    array[i] = i/10;
    array[i] = i/10;
}
var len = array.length;
while (array[i]) {
    sum += array[i];
    i++;
}
```

<Figure 6: Example of inconsistent object layouts.>

```javascript
function sum(base, p1, p2) {
    return base[p1] + base[p2];
}
for(var i=1;i<1000000;i++) {
    sum(new C(i), 'a', 'b');
}
```

<Figure 7: State machine of an array.>

![State machine of an array](image)

UNK means uninitialized array of unknown type.
NUM means numeric array. NON means non-numeric array.
Figure 6, runs 7.5% and 19.9% faster in Firefox and Chrome, respectively.\footnote{All performance improvements reported in this paper are statistically significant; Section 4.1 explains our methodology in detail.}

**Explanation** The reason for this speedup is that the original code creates objects with two possible layouts of the properties. In one layout, a appears at offset 0 and b appears at offset 1, whereas in the other layout, the order is reversed. As a result, the JIT compiler fails to specialize the code for the property lookups in `sum`. Instead of accessing the properties at a fixed offset, the executed code accesses the properties via an expensive hash table lookup. We refer to [15] for a detailed explanation of the problem.

**Profiling** To find performance problems caused by inconsistent object layouts, Profiler IOL in Table 2 tracks the hidden class associated with each object and uses the unfriendliness counter to store the number of inline cache misses that occur at code locations that access properties. The profiler implements the `newObject()` and `putProp()` functions to create or update the profiler’s representation of the hidden class of an object. This representation abstract from the implementation of hidden classes in JavaScript engines by representing the class as a list of the object’s property names, in the order in which the object’s properties are initialized. The `getOrCreateHC()` function (in Table 2) iterates over the property names of the object and checks if there exists a hidden class that matches the list of property names. If there is a matching hidden class, the function returns this hidden class, and the profiler associates it with the object. Otherwise, the profiler creates a new list of property names and associates it with the object. The profiler also caches created hidden classes for later reuse.

Based on the hidden class information, the profiler tracks whether property accesses cause inline cache misses by maintaining the following shadow-information for each location with a put or get property operation: (i) The `cachedHC` storage, which points to the hidden class of the most recently accessed base object. (ii) The `cachedProp` storage, which stores the name of the most recently accessed property. Whenever the program performs a get or put property operation, the profiler updates the information associated with the operation’s code location. If the hidden class of the operation’s base object or the accessed property differs from `cachedHC` and `cachedProp`, respectively, then the profiler increments the unfriendliness counter. This case corresponds to an inline cache miss, i.e., the JIT compiler cannot execute the code specialized for this location and must fall back on slower, generic code. At the end of the execution, the profiler reports code locations with a non-zero unfriendliness counter and ranks them in the same way as described in Section 2.2.1.

For the example in Figure 6, JITProf identifies two inline cache misses at line 11, and reports the following message:

```
Prop. access at line 11:10 has missed cache 99999 time(s)
Accessed "a" of obj. created at line 14:11 99999 time(s)
Layout [\{b\}a]: Observed 50000 time(s)
Layout [\{a\}b]: Observed 49999 time(s)
```

2

**Explanation** The code on the left of Figure 8 creates a large generic array and stores integer values ranging between 0 and 254 into it. Modifying the code so that it uses the typed array `Uint8Array`, improves performance by 60.1% and 29.6% in Firefox and Chrome, respectively.

**Profiling** To detect performance problems caused by unnecessary use of generic arrays, Profiler GA in Table 2 tracks operations performed on such arrays. The profiler associates the following boolean flags with each generic array; each flag represents a reason why a generic array cannot be replaced by a typed array: (i) One flag per kind of typed array, which represent whether the array stores elements that cannot be stored into the particular typed array. For example, `array[1] = 0.1` excludes all typed arrays that can store only integer values, such as `Uint8Array` and `Uint16Array`. (ii) Whether the program applies the `typeof` operator on the array. If the program checks the array’s type, changing the type may change the program’s semantics. (iii) Whether the program uses built-in functions of generic arrays, such as `array.slice()`. (iv) Whether the program uses the array like an object, e.g., by attaching a property to it. The profiler reports these arrays and a list of typed array constructors that can be used for creating the array.

Note that due to the nature of dynamic analysis, the profiler reports for this JIT-unfriendly code pattern is based on one execution and thus not sound for all execution paths. Instead, the profiler recommends potentially unnecessary uses of generic arrays and, in contrast to the other analyses, relies on the developer to determine whether or not it is safe to refactor those arrays.

For the example in Figure 8, the profiler reports the generic array creation at line 2 and suggests to use a `Uint8Array`.

Table 3 summarizes the JIT-unfriendly code patterns and the performance improvements discussed in this section. Since different JavaScript engines perform different optimizations, they suffer to a different degree from particular JIT-unfriendly code patterns. JITProf can address both engine-specific and cross-engine patterns. Most patterns we address here cause performance problems in multiple engines.
The profilers described in this section approximate the behavior of popular JIT engines to identify JIT-unfriendly code locations. These approximations are based on simplifying assumptions about how JIT compilation for JavaScript works, which may not always hold for every JavaScript engine. For example, we model inline caching in a monomorphic way and ignore the fact that a JavaScript engine may use polymorphic inline caching. Approximating the behavior of the JavaScript engine is a deliberate design decision that allows for implementing analyses for JIT-unfriendly code patterns with fewer lines of code, and without requiring knowledge about the engine’s implementation details.

### 2.3 Sampling

Profiling all runtime events that may be of interest for profilers imposes a significant runtime overhead. To enable developers to use JITProf as a practical profiling tool, we use sampling to reduce this overhead. We use both function level and instruction level sampling, combined in a decaying sampling strategy that focuses the profiling effort on locations that provide evidence for being JIT-unfriendly. During our experiments, sampling reduces the runtime overhead from a median of 627x to a median of 18x, without changing the recommendations reported to the user.

#### Function Level Sampling

JITProf transforms each function body $p_c$ of the analyzed program so that it contains both the original program code $p$ of the function body and the instrumented code $p'$ of the function body:

$$p_c = \text{function} (...) \{ \text{if (flag) } p \text{ else } p' \}$$

During the program’s execution, JITProf controls the overhead imposed by profiling the function by switches the flag to selectively run the original or the instrumented code. This level of sampling reduces the overhead caused by the added code inside the instrumented program.

#### Instruction Level Sampling

The instrumented code $p'$ invokes the functions in the upper part of Table 1 to notify profilers about runtime events. To enable fine-grained control of JITProf’s overhead, we complement function level sampling with instruction level sampling. Therefore, we maintain flag for every code location that may trigger a runtime event of interest and notify profilers only if the flag is set to true. By controlling these flags, JITProf can focus the profiling effort on locations that are of particular interest. This level of sampling additionally reduces the overhead caused by JITProf analyses.

#### Sampling Strategy

The sampling strategy decides when to enable profiling for a particular function and instruction. As a default, JITProf uses a decaying sampling strategy. Conceptually, JITProf assigns a sampling rate to each function and instruction, and takes a random decision according to the current sampling rate whenever the function or instruction is executed. The decaying sampling strategy starts by profiling all executions of a function or instruction, and then gradually reduces the sampling rate as the function or instruction is triggered more often. The sampling rate is $1/(1+n)$, where $n$ is the number of samples retrieved so far from a particular function or instruction. Once the sampling rate reaches a very low value (0.05%), we keep it at this value to allow JITProf to detect code locations as JIT-unfriendly even if their JIT unfriendliness only shows after reaching the location many times.

### 3. IMPLEMENTATION

To avoid limiting JITProf to a particular JavaScript engine, we implement it via a source-to-source transformation that adds analysis code to a given program. The implementation builds on the instrumentation and dynamic analysis framework JALANGI [40] and is available as open-source. JITProf tracks unfriendliness counters for code locations via a global map that assigns unique identifiers of code locations to the current unfriendliness counter at the location. The map is filled lazily, i.e., JITProf tracks counters only for source locations involved in a JIT-unfriendly pattern. To implement sampling, JITProf precomputes random decisions before the program’s execution to avoid the overhead of taking a random decision [27].

To be easily extensible to support further JIT-unfriendly code patterns, JITProf offers an API that has two parts. First, JITProf provides callback hooks that analyzes implement to track particular runtime operations of the program. The operations are at a lower level than JavaScript statements, e.g., complex expressions are split into multiple unary and binary operations. Second, JITProf provides an API for functionalities shared by several analyses, such as accessing the shadow value of an object, maintaining an unfriendliness counter for code locations, and ranking locations by their unfriendliness counter. Based on the JITProf infrastructure, our implementations of the analyses in Section 2 require between 56 and 385 lines of JavaScript code.

### 4. EVALUATION

We evaluate JITProf by studying the prevalence of JIT-unfriendly code in real-world JavaScript programs and by assessing its effectiveness as a profiler to detect optimization opportunities in benchmarks that are commonly used to assess JavaScript performance.

#### 4.1 Experimental Methodology

To study the prevalence of JIT-unfriendly code in the web, we apply JITProf to the 50 most popular websites. For each site, we analyze the JavaScript code executed by loading the start page and by manually exercising the site with a few typical user interactions. Furthermore, we apply JITProf to all benchmarks from the Google Octane and the SunSpider benchmarks.

To evaluate JITProf as a profiler that detects optimization opportunities, we apply it to all benchmarks and inspect the top three reported code locations per program and pattern, refactor them in a semantics-preserving way by replacing JIT-unfriendly code with JIT-friendly code, and measure whether these simple changes lead to a significant performance improvement in the Firefox and Chrome browsers. Each change fixes only the problem reported by JITProf and does not apply any other optimization.

---

To evaluate JITProf as a profiler, we focus on benchmark programs for three reasons. First, popular JavaScript engines are highly tuned towards these benchmarks, i.e., finding optimization opportunities is particularly challenging. Second, reliably measuring the performance of an interactive website is challenging, e.g., because it depends on user and network events. JSBench [38] addresses this problem by recording code executed by a website, but is not applicable in our evaluation because it radically changes the structure of the code, e.g., by unrolling loops\(^3\). Finally, refactoring the JavaScript code of websites is challenging because most JavaScript files on each of those websites are minified and uglified, and because we cannot easily change the code on the server that responds to AJAX requests.

To assess whether a change improves the performance, we compare the execution time of the original and the modified program in two popular browsers, Firefox 31.0 and Chrome 36.0. To obtain reliable performance data [14, 30, 9], we repeat the following steps 50 times: (1) Open a fresh browser instance and run the original benchmark. (2) Open a fresh browser instance and run the modified benchmark. Each run yields a benchmark score that summarizes the performance. Given these scores, we use the independent T-test (95% confidence interval) to check whether there is a statistically significant difference. Confidence intervals of improvements of Firefox and Chrome in the last two columns are at 95% confidence level [14].

All JIT-unfriendly code locations detected by JITProf and their refactorings are documented in our technical report [15]. Due to limited space, we only discuss a few representative examples in the following.

**Inconsistent Object Layouts in Octane-Splay.** JITProf reports a code location where inconsistent object layouts occur a total of 135 times. The layout of the objects at a statement in the ranked list of reported code locations. The table also shows the amount of changes to avoid the problem. The last two columns of the table show the performance improvement achieved with these changes, in Firefox and Chrome, respectively.

<table>
<thead>
<tr>
<th>Benchmark: SunSpider (SS) &amp; Octane (Oct)</th>
<th>CPR (FF/CH)</th>
<th>JITProf Rank (statement level)</th>
<th>Avg. improvement (stat. significant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS-Crypto-SHA1</td>
<td>2 ±5+</td>
<td>1 in UAE, PO, BOU</td>
<td>6.3 ±0.9%</td>
</tr>
<tr>
<td>SS-Str-Tagcloud</td>
<td>- ±5</td>
<td>1 in IOL</td>
<td>15 ±0.1%</td>
</tr>
<tr>
<td>SS-Crypto-MD5</td>
<td>3 ±5+</td>
<td>1 in UAE, PO, BOU</td>
<td>6.1 ±0.1%</td>
</tr>
<tr>
<td>SS-Format-Tofte</td>
<td>2 ±1</td>
<td>1 in UAE</td>
<td>2 ±4.0%</td>
</tr>
<tr>
<td>SS-3d-Cube</td>
<td>5 ±5</td>
<td>1 in NCA</td>
<td>1 ±1.0%</td>
</tr>
<tr>
<td>SS-Format-Xparb</td>
<td>4 ±1</td>
<td>1 in PO</td>
<td>2 ±19.7%</td>
</tr>
<tr>
<td>SS-3d-Raytrace</td>
<td>5 ±1</td>
<td>1 in NNA</td>
<td>4 ±22.4%</td>
</tr>
<tr>
<td>SS-3d-Morph</td>
<td>1 ±1</td>
<td>1 in GA</td>
<td>1 ±1.5%</td>
</tr>
<tr>
<td>SS-FlipFlop</td>
<td>1 ±1</td>
<td>1 in GA</td>
<td>3 ±5.4%</td>
</tr>
<tr>
<td>Oct-Splay</td>
<td>5 ±5</td>
<td>1 in IOL</td>
<td>2 ±15.1%</td>
</tr>
<tr>
<td>Oct-SplayLatency</td>
<td>5 ±5</td>
<td>1 in IOL</td>
<td>2 ±3.8%</td>
</tr>
<tr>
<td>Oct-DeltaBlue</td>
<td>5 ±5+</td>
<td>2 in IOL</td>
<td>6 ±1.4%</td>
</tr>
<tr>
<td>Oct-RayTrace</td>
<td>5 ±1</td>
<td>1 in IOL</td>
<td>18 ±12.9%</td>
</tr>
<tr>
<td>Oct-Box2D</td>
<td>5 ±5+</td>
<td>2 in IOL</td>
<td>1 ±7.5%</td>
</tr>
<tr>
<td>Oct-Crypto</td>
<td>5 ±5+</td>
<td>1 in GA</td>
<td>1 ±3.3%</td>
</tr>
</tbody>
</table>

CPR means CPU Profiler Rank. FF means Firebug Profiler. CH means Google Chrome’s profiler. Ch. LOC is the number of changed LOC. Short names (e.g., IOL) in the third column refers to the profilers defined in Table 2. - means no ranking or no statistically significant difference. Confidence intervals of improvements of Firefox and Chrome in the last two columns are at 95% confidence level [14].

\(^3\)As a result of unrolling, JITProf would miss, e.g., a JIT-unfriendly code location in a loop because each location is triggered at most once.
To avoid this problem, we refactor `String.leftPad` by replacing line 2 with:

```javascript
1 var result = val + '';
2 var tmp = new String(val) + '';
```

The modified code initializes `result` with a primitive string value. For a fair performance comparison, we add the statement at line 2 to retain a `String` object construction operation and a monomorphic "object + string" concatenation operation. This simple change leads to 19.7% and 22.4% performance improvement in Firefox and Chrome, respectively. Fixing the problem by removing the statement that calls the `String` constructor, which is the solution a developer may choose, leads to even larger speedup.

### Multiple undefined-related Problems in SunSpider-MD5

JITProf reports occurrences of three JIT-unfriendly code patterns for the following code snippet:

```javascript
function str2binl(str) {
  var len = (str.length*chrsz)>>5; var bin=new Array(len);
  for (var i = 0; i < len; i++) bin[i] = 0;
  var mask = (1 << chrsz) - 1;
  for (var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
  return bin;
}
```

The function creates an empty array and reads uninitialized elements of the array in a loop before assigning values to those elements. JITProf reports that the code accesses `undefined` elements of an array 3,956 times at line 4, that this line repeatedly performs bitwise OR operations on the `undefined` value, and that this operation is polymorphic because it operates on numbers and `undefined`. This refactoring avoids these JIT-unfriendly operations:

```javascript
function str2binl(str) {
  var bin = Array(); var mask = (1 << chrsz) - 1;
  for (var i = 0; i < len; i++) bin[i] = 0;
  var len = (str.length*chrsz)>>5; var bin=new Array(len);
  for (var i = 0; i < len; i++) bin[i] = 0;
  var mask = (1 << chrsz) - 1;
  for (var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
  return bin;
}
```

The modified code initializes the array `bin` with a predefined size (stored in the variable `len`) and then initializes all of its elements with zero. Although we introduce additional code, this change leads to a 24.6% improvement in Chrome.

### Non-contiguous Arrays in SunSpider-Cube

JITProf detects code that creates a non-contiguous array 208 times. The example is similar to Figure 3: an array is initialized in reverse order, and we modify the code by initializing the array from lower to higher index. As a result, the array increases contiguously, which results in a small but statistically significant improvement of 1.1% in Chrome.

#### Comparison with CPU-Time Profiling

The most popular existing approach for finding performance bottlenecks is CPU-time profiling [17]. To compare JITProf with CPU-time profiling, we analyze the benchmark programs in Table 4.3.1 with the Firebug Profiler and Google Chrome’s CPU Profiler. CPU-time profiling reports a list of functions in which time is spent during the execution, sorted by the time spent in the function itself, i.e., without the time spent in callees. The “CPU Profiler Rank” column in Table 4.3.1 shows for each JIT-unfriendly location identified by JITProf the CPU profiling rank of the function that contains the code location. Most code locations appear on a higher rank in JITProf’s output than with CPU profiling. The function of one code location (SunSpider-String-Tagcloud) does not even appear in the Firebug Profiler’s output, presumably because the program does not spend a significant amount of time in the function that contains the JIT-unfriendly code.

In addition to the higher rank of JIT-unfriendly code locations, JITProf improves upon traditional CPU-time profiling by pinpointing a single code location and by explaining why this location causes slowdown. In contrast, CPU-time profiling suggests entire functions as optimization candidates. For example, the performance problem in SunSpider-Format-Tofte is in a function with 291 lines of code. Instead of letting developers find optimization opportunities in this function, JITProf precisely points to the problem.

Overall, our results suggest that JITProf enables developers to find JIT-unfriendly code locations quicker than CPU-time profiling. In practice, we expect both JITProf and traditional CPU-time profiling to be used in combination.

Developers can identify JIT compilation-related problems quickly with JITProf and, if necessary, use other profilers afterwards.

#### 4.3.3 Non-optimizable JIT-Unfriendly Code

For some of the JIT-unfriendly code locations reported by JITProf, we fail to improve performance with a simple refactoring. A common pattern of such non-optimizable code is an object that is used as a dictionary or map. For such objects, the program initializes properties outside of the constructor, making the object structure unpredictable and leading to multiple hidden classes for a single object. Dictionary objects often cause inline cache misses because the object’s structure varies in an unpredictable way at runtime, but we cannot easily refactor such problems. Other common patterns are JIT-unfriendly code that is not executed frequently and code where eliminating the JIT-unfriendly code requires adding statements. For example, creating consistent object layouts may require adding property initialization statements in a constructor, and executing these additional statements takes more time than the time saved from avoiding the JIT-unfriendly code. Developers can avoid optimizing such code by inspecting only the top-ranked reports from JITProf, which occur relatively often.

#### 4.4 Runtime Overhead

Table 4.3.1 shows the time for profiling benchmarks and the slowdown compared to normal execution. As shown by the “Time” and “PS” columns, a naive implementation of JITProf imposes a significant runtime overhead (median: 627x). Fortunately, sampling (Section 2.3) reduces this overhead to a median of 18x, without changing the JIT-unfriendly code locations reported by JITProf. The slowdown with sampling is in the same order of magnitude as that of comparable dynamic analyses [40, 35, 25]. We consider the overhead to be acceptable during testing because both client-side and server-side JavaScript applications typically handle events within a few seconds to ensure that the application is responsive. Improving the performance of frequently executed event handlers can potentially lead to better user experience.
experience in the browser and increased throughput of the server. Besides sampling, our implementation is not particularly optimized for performance but instead focuses on providing a JavaScript engine-independent and easily extensible framework. We believe that other optimizations or more sophisticated sampling [10, 43] can reduce overhead even further.

5. RELATED WORK

Just-in-time Compilation Recent work includes trace-based dynamic type specialization [13], memoization of side effect-free methods [51], identifying and removing short-lived objects [41], just-in-time value specialization [8], and studying how the effectiveness of JIT compilation depends on the compilation order [11]. Hackett et al. [19] propose a static-dynamic type inference that allows for omitting unnecessary runtime checks. Ahn et al. modify the structure of hidden classes to increase the inline caching hit rate [4]. These approaches modify the JIT engine to improve the performance of existing programs, whereas JITProf supports developers in refactoring a program to improve its performance on existing JavaScript engines. We expect future improvements of JIT compilation, but we also believe that there will always remain JIT-unfriendly code.

Performance Analysis and Profiling Performance bugs are common [23] and various approaches detect and diagnose them. St-Amour et al. [45] modify a compiler so that it suggest code changes that enable additional optimizations; they have recently adapted the approach to JavaScript [44]. In contrast to this compile time analysis implemented inside a (JIT) compiler, JITProf is a runtime analysis that is implemented without modifying the JavaScript engine, making it easier for non-experts to support additional JIT-unfriendly code patterns. JITInspector shows which optimizations the JIT compiler optimizes and an intermediate representation of the generated code. The approach seems appropriate to debug a JIT compiler. JITProf targets developers of JavaScript programs. Developers compare the execution time of code snippets across JavaScript engines and get advice on how to write efficient code [54]. In contrast to these generic and program-agnostic guidelines, our approach pinpoints program-specific optimization opportunities.

Other work profiles the interaction between a program and its execution environment, e.g., regarding memory caching [12],

energy consumption [26, 6], and other interactions [22] Perf-Diff helps localize performance differences between execution environments [55]. Instead, JITProf pinpoints problems that may exist in multiple execution environments.

Xu et al. and Yan et al. propose approaches to find excessive memory usage [49, 50, 52], TAEDS is a framework to record and analyze data structure evolution during the execution [46]. Marinov and O’Callahan propose an analysis to find optimization opportunities due to equal objects [28], and Xu refines this idea to detect allocation sites where similar objects are created repeatedly [47]. Toddler [33] detects loops where many iterations have similar memory access patterns. Hammacher et al. propose a dynamic analysis to identify potential for parallelization [20]. Profiling is also used to understand the performance of interactive user interface applications [24, 37, 34] and large-scale, parallel HPC programs [5, 42]. Other approaches combine traces from multiple users to localize performance problems [21, 53]. In contrast to all the above, JITProf detects performance problems specific to the program’s execution environment.

JavaScript Analysis Dynamic analysis for JavaScript is useful for lint-like checking of coding rules [16], to find inconsistent types [35], and to empirically study the usage of particular language features [36]. JITProf complements these approaches by addressing performance problems related to JIT compilation.

6. CONCLUSION

This paper presents JITProf, a profiling framework to pinpoint code locations that prohibit profitable JIT optimizations. We instantiate the framework for seven code patterns that lead to performance bottlenecks on popular JavaScript engines and show that these patterns occur in popular websites, that JITProf finds instances of these patterns in widely used benchmark programs, and that simple changes of the programs to avoid the JIT-unfriendly code lead to significant performance improvements. Given the increasing popularity of JavaScript, we consider our work to be an important step toward improving the efficiency of an increasingly large fraction of all executed software.

7. ACKNOWLEDGEMENT

This research is supported in part by NSF Grants CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, by gifts from Mozilla and Samsung, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the German Research Foundation (DFG) within the Emmy Noether Project “ConCys”. We thank Luca Della Toffola and the anonymous reviewers for their valuable feedback.
8. REFERENCES


   http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/.


