JITProf: Pinpointing JIT-Unfriendly JavaScript Code

Liang Gong[1], Michael Pradel[2], and Koushik Sen[1]

[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

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786831

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 analyzed benchmarks.

To reduce the runtime overhead that a naive implementation of JITProf imposes, we present a sampling technique that dynamically adapts the profiling effort for particular functions and instructions. With sampling, we reduce the overhead of JITProf from an average of 627x to an average overhead of 18x, while still finding all optimization opportunities that are detected without sampling.

In summary, this paper contributes the following:

- We present JITProf, an engine-independent and extensible framework that analyzes runtime information to pinpoint code locations that reduce performance because they prohibit effective JIT optimization.
- 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 enables 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 *incrCtr*() 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).

Table 1: The runtime event predicates captured and analyzed in JITProf.

Function	Description
Functions that profilers can implement:	
$newObj(l, v)$	A new object v is created at location l
$getProp(l, base, p, result)$	Value $results$ is read from property p of object $base$ at location l
$putProp(l, base, p, v)$	Value v is written to property p of object $base$ at location l
$unary(l, op, v, result)$	Unary operation op applied to value v yields value $results$ at location l
$binary(l, op, v1, v2, result)$	Binary operation op applied to values $v1$ and $v2$ yields value $results$ at location l
$invokeFun(l, f, base, args, result)$	Function f of object $base$ is called with arguments $args$ and yields value $results$ at location l
Functions that profilers can call:	
$v.meta$	Shadow-information associated with runtime value v
$l.meta$	Shadow-information associated with location l
$incrCtr(l)$	Increment the unfriendliness counter of location l

```
1 function f(a,b){return a+b;}
2
3 for(var i=0;i<5000000;i++){
4   var arg1, arg2;
5   if (i % 2 === 0) {
6     a = 1; b = 2;
7
8   } else {
9     a = 'a'; b = 'b';
10
11  }
12  f(a, b);
13 }
```

```
function f(a,b){return a+b;}
function g(a,b){return a+b;}
for(var i=0;i<5000000;i++){
  var arg1, arg2;
  if (i % 2 === 0) {
    a = 1; b = 2;
    f(a, b);
  }else {
    b = 'a'; b = 'b';
    g(a, b);
  }

}
```

The highlighted code on the left pinpoints the JIT-unfriendly code location. The highlighted code on the right shows the difference of the improved code to the code on the left.

Figure 1: Example of polymorphic operation (left) and improved code (right).

2.2 Patterns and Profilers

This section describes JIT-unfriendly code patterns and the detection of their occurrences by instantiating the JITProf framework. Table 2 summarizes the profilers that detect these patterns.

2.2.1 *Polymorphic Operations*

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 `f` into a 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

```
1 var x, y, rep=300000000;
2 for(var i=0;i<rep;i++){
3   y = x | 2;
4 }
```

```
var x = 0, y, rep=300000000;
for(var i=0;i<rep;i++){
  y = x | 2;
}
```

Figure 2: Example for a binary operation on `undefined` (left) and improved code (right).

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 C_2) 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 C_2 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.

2.2.2 *Binary Operation on* `undefined`

Performing binary operations, such as `+`, `-`, `*`, `/`, `%`, `|`, and `&` on `undefined` values (which has well-defined semantics in JavaScript), degrades performance compared to applying the same operations on defined values.

Micro-benchmark The code on the left of Figure 2 reads the `undefined` value from `x` and implicitly converts it into zero. Modifying this code so that `x` is initialized to zero preserves the semantics and improves the performance by 1.8% and 82.8% in Firefox and Chrome, respectively.

Explanation The original code prevents the JavaScript engine from executing code specialized for numbers. Instead, the engine falls back on code that performs additional runtime checks and that coerces `undefined` into a number.

Profiling To detect performance problems caused by binary operations with `undefined` operands, Profiler BOU in Table 2 tracks all binary operations and increments the unfriendliness counter whenever an operation operates on an `undefined` operand.

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

Runtime event	Action
(PO) Polymorphic Operations:	
$unary(l, *, v, *)$	if $(type(v) \neq l.meta.lastType)$ then $incrCtr(l)$ $l.meta.lastType \leftarrow type(v)$ $l.meta.histo.add(type(v))$
$binary(l, *, v1, v2, *)$	if $(type(v1) \neq l.meta.lastType1 \vee type(v2) \neq l.meta.lastType2)$ then $incrCtr(l)$ $l.meta.lastType1 \leftarrow type(v1)$ $l.meta.lastType2 \leftarrow type(v2)$ $l.meta.histo.add(type(v1), type(v2))$
(BOU) Binary Operations on `undefined`:	
$binary(l, *, v1, v2, *)$	if $v1 = undefined \vee v2 = undefined$ then $incrCtr(l)$
(NCA) Non-contiguous Arrays:	
$putProp(l, base, prop, *)$	if $isArray(base) \wedge isNumber(prop) \wedge (prop < 0 \vee prop > base.length)$ then $incrCtr(l)$
(UAE) Accessing Undefined Array Elements:	
$getProp(l, base, prop, *)$	if $(isArray(base) \wedge isNumber(prop) \wedge prop \notin base$ then $incrCtr(l)$
(NNA) Storing Non-numeric Values in Numeric Arrays:	
$newObject(l, v)$	if $isArray(v)$ then if $containsNonNumeric(v)$ then $v.meta.state \leftarrow NON$ else if $allNumeric(v)$ then $v.meta.state \leftarrow NUM$ else $v.meta.state \leftarrow UNK$
$putProp(l, base, prop, v)$	$oldState \leftarrow l.meta$ $updateState(base.meta, v)$ if $oldState = NUM \wedge base.meta.state = NON$ then $incrCtr(l)$
(IOL) Inconsistent Object Layouts ("HC" means hidden class):	
$newObject(l, v)$	$v.meta \leftarrow getOrCreateHC(v)$
$putProp(l, base, p, v)$	$base.meta \leftarrow getOrCreateHC(v)$ if $base.meta \neq l.meta.cachedHC \vee p \neq l.meta.cachedProp$ then $incrCtr(l)$ $l.meta.cachedHC \leftarrow base.meta$ $l.meta.cachedProp \leftarrow p$ $l.meta.histo.add(base.meta)$
$getProp(l, base, p, *)$	if $base.meta \neq l.meta.cachedHC \vee p \neq l.meta.cachedProp$ then $incrCtr(l)$ $l.meta.cachedHC \leftarrow base.meta$ $l.meta.cachedProp \leftarrow p$ $l.meta.histo.add(base.meta)$
(GA) Unnecessary Use of Generic Arrays:	
$newObject(l, v)$	if $isArray(v)$ then $v.meta \leftarrow initArrayMetaInfo()$
$putProp(*, base, prop, v)$	if $isArray(base)$ then $updateTypeFlags(base.meta, prop, v)$
$unary(*, op, v, *)$	if $isArray(v) \wedge op = "typeof"$ then $setFlag(v.meta, "typeof")$
$invokeFun(*, f, base, *, *)$	if $isArray(base)$ then $setBuiltinsUsed(v.meta, f)$

```
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 }
```

```
for (var j=0; j<400; j++) {
  var array = [];
  for(var i=0;i<=5000;i++){
    array[i] = i;
  }
}
```

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 makes an array non-contiguous. For each put property operation where the base is an array and where the property is an index, the profiler checks whether the index is less than 0 or greater than the length of the array. In this case, the operation inserts an element that makes the array non-contiguous and the profiler increments the unfriendliness counter.

For the example in Figure 3, the profiler warns about line 4 because it transforms the array into a non-contiguous array every time the line is executed.

```
1 var array = [], sum = 0;
2 for(var i=0;i<100;i++)
3   array[i] = 1;
4 for(var j=0;j<100000;j++) {
5   var ij = 0;
6   var len = array.length;
7   while (array[ij]) {
8     sum += array[ij]
9     ij++;
10  }
11 }
```

```
var array = [], sum = 0;
for(var i=0;i<100;i++)
  array[i] = 1;
for(var j=0;j<100000;j++) {
  var ij = 0;
  var len = array.length;
  while (ij < len) {
    sum += array[ij];
    ij++;
  }
}
```

Figure 4: Accessing undefined array elements.

```
1 var array = [];
2 for(var i=0;i<10000000;i++)
3   array[i] = i/10;
4 array[4] = "abc";
5 array[4] = 1.23;
```

```
var array = [];
for(var i=0;i<10000000;i++)
  array[i] = i/10;
array[4] = 3;
array[4] = 1.23;
```

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

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 element, 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.

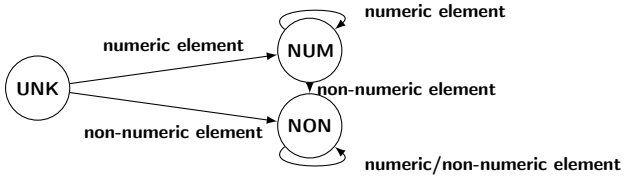
Explanation If a dense array contains only numeric values, such as 31-bit signed integers[1] or doubles, then the JavaScript engine can represent the array efficiently as a fixed sized C-like array of integers or doubles, respectively. Changing the representation of the array from a fixed-sized integer/double array to an array of non-numeric values is an expensive operation.

[1]Both the Firefox and the Chrome JavaScript engine use tagged integers [7], where 31 bits represent a signed integer and the remaining bit indicates its type (integer or pointer).

```
1 function C(i) {
2   if (i % 2 === 0) {
3     this.a = Math.random();
4     this.b = Math.random();
5   } else {
6     this.b = Math.random();
7     this.a = Math.random();
8   }
9 }
10 function sum(base, p1, p2) {
11   return base[p1]+base[p2];
12 }
13 for(var i=1;i<100000;i++) {
14   sum(new C(i), 'a', 'b');
15 }
```

```
function C(i) {
  if (i % 2 === 0) {
    this.a = Math.random();
    this.b = Math.random();
  } else {
    this.a = Math.random();
    this.b = Math.random();
  }
}
function sum(base, p1, p2) {
  return base[p1] + base[p2];
}
for(var i=1;i<100000;i++) {
  sum(new C(i), 'a', 'b');
}
```

Figure 6: Example of inconsistent object layouts.



UNK means uninitialized array of unknown type.
NUM means numeric array. **NON** means non-numeric array.

Figure 7: State machine of an array.

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 with three states: *unknown*, *numeric*, and *non-numeric* (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 6, runs 7.5% and 19.9% faster in Firefox and Chrome, respectively.[2]

Explanation The reason for this speedup is that the original code creates `C` 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)
Prop. access at line 11:21 has missed cache 99999 time(s)
  Accessed "b" 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]All performance improvements reported in this paper are statistically significant; Section 4.1 explains our methodology in detail.

```
1 var size = 5000000;
2 var arr=new Array(size);
3 for (var i=0;i<size;i++)
4   arr[i%size] = i%255;
```

```
var size = 5000000;
var arr=new Uint8Array(size);
for (var i=0;i<size;i++)
  arr[i%size] = i%255;
```

Figure 8: Inappropriate use of generic arrays.

2.2.7 *Unnecessary Use of Generic Arrays*

JavaScript has *generic arrays*, created with `new Array()` or a literal, and *typed arrays*, created, e.g., with `Int8Array()`. Typed arrays enable various optimizations and programmers should use them to improve performance.

Micro-benchmark 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.

Explanation Typed arrays allow the JIT engine to use C-like type-specialized arrays of fixed length, instead of more complex data structures. The change in Figure 8 leads to a more compact memory representation and avoids unnecessary runtime checks. JIT engines might optimize generic numeric arrays in a similar way (Section 2.2.5), but often fail to pick the most efficient array representation. Explicitly using typed arrays helps the engine optimize the program.

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 updates these flags by implementing the *putProp*(), *unary*(), and *invokeFun*() functions. At the end of the execution, the profiler identifies arrays where at least one flag from category (i) and all flags (ii) to (iv) are true. 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 result 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.

Table 3: Performance improvements on micro-benchmarks of JIT-unfriendly code patterns.

JIT-unfriendly code pattern	Firefox	Chrome
Inconsistent object layouts	7.5%	19.9%
Polymorphic operations	92.1%	72.2%
Binary operations on undefined	1.8%	82.8%
Non-contiguous arrays	97.5%	90.2%
Accessing undefined array elements	73.9%	70.2%
Storing non-numeric values in numeric arrays	14.9%	83.8%
Unnecessary use of generic arrays	60.1%	29.6%

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 a few 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:

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

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 analyses 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.[3] 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.

[3] http://www.alexa.com/

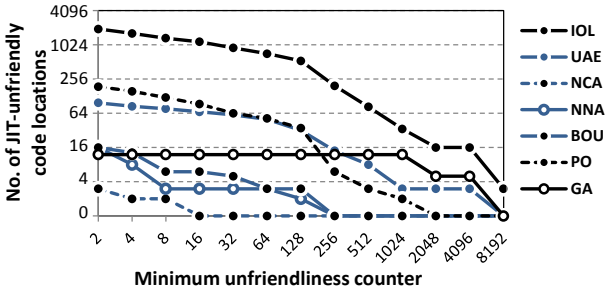


Figure 9: Prevalence of JIT-unfriendly code.

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[4]. Finally, refactoring the JavaScript code of websites is challenging because most JavasScript files on each of those website 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 performance difference between the original and the modified program. All performance differences are statistically significant. Experiments are performed on Mac OS X 10.9 using a 2.40GHz Intel Core i7-3635QM CPU machine with 8GB memory.

4.2 Prevalence of JIT Unfriendliness

Figure 9 illustrates the prevalence of JIT-unfriendly code patterns on the 50 most popular websites. The figure shows the total number of JIT-unfriendly code locations reported by JITProf (vertical axis), depending on the minimum unfriendliness counter required to consider a location as JIT-unfriendly (horizontal axis). The results show that JIT-unfriendly code patterns are prevalent in practice and that some patterns are more common than others. These results provide guidance on which patterns to focus on.

4.3 Profiling JIT-Unfriendly Code Locations

4.3.1 *JIT-Unfriendly Code Found by JITProf*

JITProf detects JIT-unfriendly code that causes easy to avoid performance problems in 15 of the 39 benchmarks. Table 4.3.1 summarizes the performance improvements achieved by avoiding these problems. The "JITProf Rank" column indicates which analysis detects a problem and the position of the problem 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.

[4]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.

Table 4: Performance improvement achieved by avoiding JIT-unfriendly code patterns.

Benchmark: SunSpider (SS) & Octane (Oct)	CPR FF|CH (function level)	JITProf Rank (statement level)	Ch. LOC	Avg. improvement (stat. significant)	
				Firefox	Chrome
SS-Crypto-SHA1	2 |5+	1 in UAE, PO, BOU	6	3.3±0.9%	26.3±0.4%
SS-Str-Tagcloud	- | 5	1 in IOL	15	-	11.7±0.7%
SS-Crypto-MD5	3 |5+	1 in UAE, PO, BOU	6	-	24.6±0.1%
SS-Format-Tofte	2 | 1	1 in UAE	2	-	3.4±0.2%
SS-3d-Cube	5+| 5	1 in NCA	1	-	1.1±0.1%
SS-Format-Xparb	4 | 1	1 in PO	2	19.7±0.5%	22.4±0.3%
SS-3d-Raytrace	5 | 5	1 in NNA	4	-	2.6±0.2%
SS-3d-Morph	1 | 1	1 in GA	1	-	1.5±0.3%
SS-Fannkuch	1 | 1	1 in GA	3	8.3±0.9%	5.4±2.3%
Oct-Splay	5 |5+	1 in IOL	2	3.5±0.9%	15.1±0.3%
Oct-SplayLatency	5 |5+	1 in IOL	2	-	3.8±0.6%
Oct-DeltaBlue	5+|5+	2 in IOL	6	1.4±0.2%	-
Oct-RayTrace	5+| 1	1 in IOL	18	-	12.9±1.9%
Oct-Box2D	5+|5+	2 in IOL	1	-	7.5±0.6%
Oct-Crypto	5+|5+	1 in GA	1	13.8±4.9%	3.3±0.4%

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].

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 that retrieves a property frequently alternate between `key|value|left|right` and `key|value|right|left`. The problem boils down to the following code, which initializes the properties `left` and `right` in two possible orders:

```
1 var node = new SplayTree.Node(key, value);
2 if (key > this.root_.key) {
3   node.left = this.root_;
4   node.right = this.root_.right;
5   ...
6 } else {
7   node.right = this.root_;
8   node.left = this.root_.left;
9   ...
10 }
```

We swap the first two statements in the `else` branch so that the object layout is always `key|value|left|right`, which improves performance by 3.5% and 15.1% in Firefox and Chrome, respectively.

Polymorphic Operations in SS-Format-Xparb. JITProf reports a code location that frequently performs a polymorphic plus operation. Specifically, the analysis observes operand types `string + string` 699 times and operand types `object + string` 3,331 times. The behavior is due to the following function, which returns either a primitive string value or a `String` object, depending on the value of `val`:

```
1 String.leftPad = function (val, size, ch) {
2   var result = new String(val);
```

```
3   if (ch == null) { ch = " "; }
4   while (result.length < size){
5     result = ch + result;
6   }
7   return result;
8 }
```

To avoid this problem, we refactor `String.leftPad` by replacing line 2 with:

```
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:

```
1 function str2binl(str) {
2   var bin = Array(); var mask = (1 << chrsz) - 1;
3   for (var i = 0; i < str.length * chrsz; i += chrsz)
4     bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
5   return bin;
6 }
```

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:

```
1 function str2binl(str) {
2   var len = (str.length*chrsz)>>5; var bin=new Array(len);
3   for (var i = 0; i < len; i++) bin[i] = 0;
4   var mask = (1 << chrsz) - 1;
5   for (var i = 0; i < str.length * chrsz; i += chrsz)
6     bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
7   return bin;
8 }
```

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.

4.3.2 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[5] 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

[5] https://getfirebug.com/wiki/index.php/Profiler

Table 5: Benchmarks used for the evaluation and performance statistics.

Benchmark	LOC	Time	PS	$\tilde{Time}$	$\tilde{PS}$	Benchmark	LOC	Time	PS	$\tilde{Time}$	$\tilde{PS}$	Benchmark	LOC	Time	PS	$\tilde{Time}$	$\tilde{PS}$
SS-Controlflow-Recursive	25	2.93	674	0.07	17	SS-String-Fasta	90	4.13	391	0.48	45	Oct-Splay	395	0.59	117	0.06	12
SS-Bitops-Bits-in-Byte	26	5.38	1520	0.13	36	SS-Math-Cordic	101	5.6	943	0.12	20	Oct-Navi-Stokes	407	41.64	1859	2.01	90
SS-Bitops-Bitwise-And	31	3.09	936	0.23	71	SS-String-Base64	136	4.16	457	0.42	46	Oct-Richards	537	2.47	386	0.12	18
SS-Math-Partial-Sums	33	3.39	301	0.25	22	SS-Access-Nbody	170	12.38	1649	0.10	13	Oct-DeltaBlue	880	3.94	267	0.24	16
SS-Bitops-Nsieve-Bits	35	7.05	920	0.33	43	SS-Crypto-SHA1	225	2.87	262	0.20	19	Oct-Raytrace	904	13.45	652	0.34	16
SS-Bitops-3bit-Bits	38	4.12	1577	0.16	62	SS-String-Tagcloud	266	4.88	173	0.36	13	Oct-Code-Load	1527	2.08	108	0.3	16
SS-Access-Nsieve	39	3.51	585	0.33	55	SS-Crypto-MD5	288	2.83	414	0.16	24	Oct-Crypto	1699	64.52	3418	0.37	20
SS-Math-Spectral-Norm	51	6.13	1065	0.1	18	SS-Date-Tofte	300	9.65	652	0.15	10	Oct-Regexp	1765	6.82	91	0.7	9
SS-Access-Binary-Trees	52	4.48	1077	0.14	33	SS-3d-Cube	339	18.5	1500	0.16	13	Oct-Earl-Boyer	4683	38.73	970	0.91	23
SS-3d-Morph	56	6.48	677	0.36	37	SS-Date-Xparb	418	2.92	195	0.13	9	Oct-Box2d	9537	85.41	460	2.41	13
SS-String-Unpack-Code	67	3.09	114	0.17	6	SS-Crypto-AES	425	8.64	816	0.15	14	Oct-Gbemu	11106	294.38	1228	9.59	40
SS-Access-Fannkuch	68	11.64	1455	0.19	24	SS-3d-Raytrace	443	9.03	627	0.21	14	Oct-Typescript	25911	785.64	525	13.53	9
SS-String-Validate-Input	90	0.15	85	0.01	8	SS-Regexp-DNA	1714	0.15	14	0.02	2	Oct-Pdfjs	33071	75.16	300	5.62	22

Time means total running and analysis time JITProf (seconds). PS means profiling slowdown (×). $\tilde{Time}$ and $\tilde{PS}$ are with sampling. SS- and Oct- mean SunSpider and Octane benchmark, respectively.

experience in the browser[6] 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[7] shows which operations 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[8] 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] PerfDiff 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, 48, 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 "ConcSys". We thank Luca Della Toffola and the anonymous reviewers for their valuable feedback.

[6]Studies show that over 0.1s delay in responsive UI causes the user to feel disconnected from the interface [29, 31].

[7]https://addons.mozilla.org/en-US/firefox/addon/jit-inspector/

[8]http://jsperf.com

8. REFERENCES

[1] Native vs Typed JS Array Speed (last visited: May 2015). http://jsperf.com/native-vs-typed-js-array-speed/23.

[2] Performance Tips for JavaScript in V8 (last visited: May 2015). http://www.html5rocks.com/en/tutorials/speed/v8/.

[3] Writing Fast, Memory-Efficient JavaScript (last visited: May 2015). http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/.

[4] W. Ahn, J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas. Improving JavaScript performance by deconstructing the type system. In *PLDI*, 2014.

[5] R. Bell, A. D. Malony, and S. Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par*, pages 17–26, 2003.

[6] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, pages 169–180. ACM, 2005.

[7] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self - a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70, 1989.

[8] I. Costa, P. Alves, H. N. Santos, and F. M. Q. Pereira. Just-in-time value specialization. In *CGO*, pages 1–11, 2013.

[9] C. Curtsinger and E. D. Berger. STABILIZER: statistically sound performance evaluation. In *ASPLOS*, pages 219–228. ACM, 2013.

[10] M. Diep, M. B. Cohen, and S. G. Elbaum. Probe distribution techniques to profile events in deployed software. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, pages 331–342. IEEE Computer Society, 2006.

[11] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. Finding the limit: examining the potential and complexity of compilation scheduling for jit-based runtime systems. In *ASPLOS*, pages 607–622, 2014.

[12] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA*, pages 2–10. IEEE Computer Society, 1999.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478, 2009.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76. ACM, 2007.

[15] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. Technical Report UCB/EECS-2014-144, EECS Department, University of California, Berkeley, Aug 2014.

[16] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *ISSTA*, 2015.

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[18] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.

[19] B. Hackett and S. yu Guo. Fast and precise hybrid type inference for javascript. In *PLDI*, pages 239–250. ACM, 2012.

[20] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 49–55, Washington, DC, USA, 2009. IEEE Computer Society.

[21] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155. IEEE, 2012.

[22] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 251–269, 2004.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[24] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–170. ACM, 2011.

[25] E. Lavoie, B. Dufour, and M. Feeley. Portable and efficient run-time monitoring of javascript applications using virtual machine layering. In *ECOOP*, pages 541–566, 2014.

[26] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[27] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 141–154. ACM, 2003.

[28] D. Marinov and R. O'Callahan. Object equality profiling. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[29] R. B. Miller. Response time in man-computer conversational transactions. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I*, volume 33 of *AFIPS Conference Proceedings*, pages 267–277. AFIPS / ACM / Thomson Book Company, Washington D.C., 1968.

[30] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276, 2009.

[31] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[32] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[33] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571. IEEE / ACM, 2013.

[34] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[35] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*, 2015.

[36] M. Pradel and K. Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in javascript. In *ECOOP*, 2015.

[37] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Conference on Operating Systems Design and Implementation (OSDI)*, pages 107–120. USENIX, 2012.

[38] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, pages 677–694, 2011.

[39] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, pages 1–12. ACM, 2010.

[40] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[41] A. Shankar, M. Arnold, and R. Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142. ACM, 2008.

[42] S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, pages 287–311, 2006.

[43] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 561–578. ACM, 2014.

[44] V. St-Amour and S. Guo. Optimization coaching for javascript. In *ECOOP*, 2015.

[45] V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *OOPSLA*, pages 163–178, 2012.

[46] X. Xiao, J. Zhou, and C. Zhang. Tracking data structures for postmortem analysis. In *ICSE*, pages 896–899. ACM, 2011.

[47] G. Xu. Finding reusable data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034. ACM, 2012.

[48] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173. ACM, 2010.

[49] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430. ACM, 2009.

[50] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[51] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 75–82. ACM, 2007.

[52] D. Yan, G. H. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering, (ICSE)*, pages 134–144. IEEE, 2012.

[53] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: a device-driver case. In *ASPLOS*, pages 193–206, 2014.

[54] N. C. Zakas. High performance javascript - build faster web application interfaces. 2010.

[55] X. Zhuang, S. Kim, M. J. Serrano, and J.-D. Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Symposium on Code Generation and Optimization (CGO)*, pages 4–13. ACM, 2008.