JITProf: Pinpointing JIT-unfriendly JavaScript Code

Liang Gong
Michael Pradel
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-144
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-144.html

August 3, 2014

JITPROF: Pinpointing JIT-unfriendly JavaScript Code

Liang Gong Michael Pradel Koushik Sen
{gongliang13,pradel,ksen}@cs.berkeley.edu
EECS Department, UC Berkeley

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 technique that helps developers to identify such JIT-unfriendly *code. This paper presents* JITPROF*, a profiling framework to dynamically identify code locations that prohibit profitable JIT optimizations. The basic idea is to associate execution counters with potentially JIT-unfriendly code locations and to use these counters to report code locations that match code patterns known to prohibit optimizations. We instantiate the idea for six JIT-unfriendly code patterns that cause performance problems in the Firefox and Chrome browsers, and we apply the approach to popular benchmark programs. Our results show that refactoring these programs to avoid performance problems identified by* JITPROF *leads to performance improvements of up to 26.3% in 12 benchmarks.*

1. Introduction

JavaScript is the most widely used client-side language for writing web applications. It powers various popular websites including Gmail, Facebook, Twitter, and Google Docs. More recently, JavaScript has found its way into mobile platforms, such as Firefox OS, Tizen OS, iOS, and Android, as well as desktop platforms, such as Windows 8 and Chrome OS. A key reason behind the popularity of JavaScript is that it can run on any platform that has a modern web browser.

Another key reason behind the popularity of JavaScript among programmers is that it is a flexible, dynamically typed language. For example, programmers can add and delete object properties, and truncate and expand arrays at any point during an execution. Access to nonexistent object properties and out of bounds array elements returns the `undefined` value instead of raising an exception. These dynamic features make programming in JavaScript convenient, but they also make it difficult to compile JavaScript into efficient machine code. For example, the fact that object properties can be added and deleted forces JavaScript engines to implement objects as hash tables, making property lookup an expensive operation.

To avoid these performance penalties, modern JavaScript engines implement just-in-time (JIT) compilation [3, 8, 13, 16], which translates and optimizes JavaScript code into efficient machine code while the program executes. An important premise for effective JIT optimization is that programmers use the dynamic features of JavaScript in a systematic way. For example, JIT compilers exploit the fact that object properties are often added to an object of a given type in a specific order or that out of bounds array accesses occur rarely. JIT compilers exploit these regularity assumptions to generate efficient machine code at runtime. If a code block satisfies the assumptions, the JavaScript engine executes efficient, generated machine code. Otherwise, the engine must fall back to slower code or to interpreting the program.

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 JIT optimizations. We call such code *JIT-unfriendly* code. Previous research [32] shows that programmers extensively use the dynamic features of JavaScript, including dynamic addition and deletion of object properties, that often result in JIT-unfriendly code. Unfortunately, there currently is no technique that helps developers to identify JIT-unfriendly code.

In this paper, we propose a profiling framework, called JITPROF, that dynamically identifies code locations that prohibit profitable JIT optimizations. The basic idea is to associate execution counters to potentially JIT-unfriendly operations and to use these counters to report code locations that match code patterns known to be JIT-unfriendly. JITPROF associates meta-information with JavaScript objects, updates this information at runtime, 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.

We implement JITPROF[1] in a prototype framework written purely in JavaScript. JITPROF instruments JavaScript code through source-to-source transformation and the instrumented code checks and reports at runtime various JIT-unfriendly code locations. We instantiate the JITPROF framework for six JIT-unfriendly code patterns that cause performance problems in the Firefox and Chrome browsers. We apply our approach to the programs in the SunSpider and Octane benchmark suites. Even though the JavaScript engines of Firefox and Chrome

[1] The tool is available as open-source at https://github.com/Berkeley-Correctness-Group/Jalangi-Berkeley/tree/master/src/js/analyses/jitaware

are tuned towards these benchmarks, JITPROF identifies various JIT-unfriendly code locations. Based on these reports, we manually refactor 11 programs by replacing JIT-unfriendly code with JIT-friendly code. These simple changes give statistically significant improvements of execution time of up to 19.7% and 26.3% in Firefox and Chrome, respectively. In the ranked list of code locations that JITPROF reports, all of these optimization opportunities are at the first or second position. In contrast, traditional CPU-time profiling often reports the methods that contain these code locations at a lower rank.

A key advantage of JITPROF compared to traditional CPU-time profilers [15] is that it not only identifies performance problems, but it also explains the cause of these problems. This is particularly important for scripting languages whose performance behaviors are often unpredictable and heavily depend on the runtime characteristics of the program.

Another key advantage of JITPROF is that the framework can easily be extended to incorporate other programming patterns that prevent JIT optimizations. Such an extension does not require detailed knowledge of the internals of a JIT compiler. Instead, an extension writer needs to understand the JIT-unfriendly code pattern at a high-level and to write a few lines of JavaScript code that use JITPROF's API. Our existing analyses require between 53 and 278 lines of JavaScript code. This user-level extensibility is important because JIT compilers evolve rapidly and because different JIT compilers may not employ the same set of optimizations.

A third advantage of JITPROF is that it is not tied to any JavaScript engine. Our prototype tool is written purely in JavaScript. As an alternative, a JITPROF-like approach could be implemented within a JavaScript engine. However, such an implementation would be hard to maintain and extend by anyone who is not familiar with the specific engine, and it would not be useful for other JIT engines that implement potentially different set of optimizations.

In summary, this paper makes the following contributions:

- We study popular JavaScript engines and extract common code patterns that reduce performance because they prohibit effective JIT optimization (Section 2).
- We present a profiling framework that automatically detects these JIT-unfriendly code patterns at runtime (Section 3) and describe its implementation as a JIT engine-independent source-to-source transformation (Section 4).
- We evaluate the approach with 40 well known JavaScript benchmark programs and demonstrate that our approach pinpoints valuable optimization opportunities in 12 of them.

2. JIT-unfriendly Code Patterns

This section presents and discusses some of the important code patterns that are difficult to handle for JavaScript JIT compilers, and that often lead to performance bottlenecks. JITPROF detects instances of all these *JIT-unfriendly code patterns* in widely used benchmark programs. In this section, we illustrate each pattern with a simple example. The full source code of these micro-benchmarks is available for download.[2]

```
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 1: Example for inconsistent object layouts (left) and improved code (right). The highlighted code on the left pinpoints the JIT-unfriendly code location. The highlighted code on the right shows the difference to the code on the left.

The code patterns described in this section are JIT-unfriendly for two reasons. First, several patterns prevent the JIT compiler from executing specialized code. For example, JIT compilers often speculate that runtime types at a code location match the types observed in prior executions of this location. Second, several patterns prevent the JIT compiler from using a specialized data representation or force the JIT compiler to transform data from one representation to another.

2.1. 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 the sets of object properties. Such *inconsistent object layouts* prevent an optimization that specializes property accesses based on recurring object layouts.

Example. The program in Figure 1 has two functions: `C` is a constructor that creates an object with two properties `a` and `b`. The properties are initialized in two possible orders depending on the value of the constructor's parameter `i`. The function `sum` has three parameters: `base` is an object, and `p1` and `p2` are property names. The expression `base[p1]` returns the value of the property whose name is stored as a string in the variable `p1`. The main loop of the program repeatedly constructs objects of type `C` and passes them to `sum`. The performance of the example can be significantly improved by swapping lines 6 and 7. The modified code, given on the right of Figure 1, runs 7.5% and 19.9% faster than the original example in Firefox and Chrome, respectively.[3]

Explanation. The reason for this speedup is that the original code creates numerous `C` objects with two possible layouts of the properties. In one layout, `a` appears at offset 0 and `b`

[2] https://github.com/Berkeley-Correctness-Group/Jalangi-Berkeley/tree/master/tests/jitaware/experiments/benchmarks/microbench

[3] All performance improvements reported in this paper are statistically significant; Section 5.1 explains our methodology in detail.

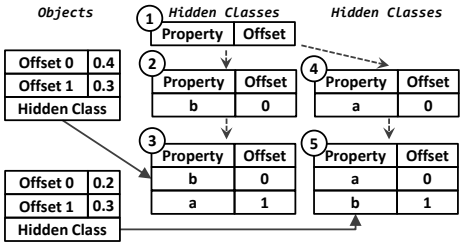


Figure 2: Structure of objects and hidden classes.

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 optimizing this code so that it accesses the properties at a fixed offset, the executed code accesses the properties via an expensive hash table lookup. We next explain this problem in more detail.

Background: Hidden Classes and Inline Caching. In JavaScript, an object is a map from property names to property values. Looking up a property of an object can be expensive if the object is represented as a hash map. To avoid expensive property lookups, most JavaScript engines store a representation of each object's memory layout and use this representation by caching the offset of properties accessed at particular code locations. For this purpose, the object is represented in two parts: an array with the values of the object's properties and a map from property names to offsets, called *hidden class*. A property lookup has two steps. First, the JavaScript engine retrieves the offset of the property from the hidden class. Second, the engine uses the offset to retrieve the value of the property from the array representing the object. Whenever a new property is added to an object, the engine appends the value to the array that represents the object and updates the object's hidden classes so that it reflects the new object layout.

Figure 2 shows how objects of type `C` from the original example in Figure 1 are represented in memory. When an object of type `C` is instantiated at line 14, an empty array representing the object is created and the hidden class of the object points to an empty map (1). When the execution reaches line 6, the array of the object is updated by adding a value at offset 0, and the hidden class of the object now points to a map that assigns `b` to 0 (2). When executing line 7, another value is added to the array at offset 1, and the hidden class of the object now points to a map that assigns `b` to 0 and `a` to 1 (3). Similarly, if the execution takes the then branch at line 2, an object is created whose array contains values at offsets 0 and 1, respectively. However, the hidden class of this object maps `a` to 0 and `b` to 1 (5), i.e., the hidden class differs from the hidden class created when taking the other branch. The JavaScript engine reuses existing hidden classes if the order of properties matches. For example, the engine uses the same hidden classes whenever it executes the then branch in line 2.

Given the above representation of an object in JavaScript, the value of a property of an object at a code location can be retrieved quickly by a mechanism called *inline caching* [10]. The basic idea is to cache the offset of the property that was previously used at a code location and to reuse this offset if the program repeatedly accesses the same property name in the same hidden class. In other words, the JIT compiler specializes the code for property accesses based on the assumption that object layouts recur at code locations. To this end, the JavaScript engine maintains for every code location that performs a property access `base.prop` two variables: `cached_hidden_class`, which points to the hidden class that was used last at this location, and `cached_offset`, which contains the offset accessed last at this location. To access the property, the engine checks whether the hidden class of `base` is the same as `cached_hidden_class`. In this case, called *inline cache hit*, the engine returns the property value stored at `cached_hidden_class[cached_offset]`. Otherwise, called *inline cache miss*, the engine looks up the property's offset in the current hidden class and caches both for the next time the code location is reached. Inline cache misses are significantly more expensive than inline cache hits. For code that accesses a property where the property name is stored in a variable, such as at line 11 of Figure 1, inline caching stores the property name in addition to the hidden class and the offset. Real-world JavaScript engines may use more sophisticated variants of inline caching than what we describe here, such as polymorphic inline caches [20] that store multiple previously seen hidden classes and offsets.

Hidden classes and inline caching explain why changing the order of initializing properties in Figure 1 leads to a significant performance improvement. If we swap lines 6 and 7, all objects of type `C` created at line 14 have the same hidden class. Therefore, all but the first accesses to the properties `a` and `b` result in an inline cache hit. In contrast, the unmodified code results in an inline cache miss at each property access in line 11 because the cached hidden class alternates between the two hidden classes described above.

2.2. Polymorphic Operations

Another common source of JIT-unfriendly behavior are code locations that apply an operation to different sets of types when reaching the location multiple times. We call such operations *polymorphic operations*.

Example. The plus operation in Figure 3 at line 2 operates on numbers when `f` is called after executing line 10 and on strings when `f` is called after executing line 13. 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 3. The modified code runs 92.1% and 72.2% faster than the original example in Firefox and Chrome, respectively.

Explanation. This change significantly improves the performance because it enables the JavaScript engine to execute specialized code for the plus operation. The change turns a polymorphic operation into two monomorphic operations, i.e., into operations that always execute on the same types of

```
1  function f(a, b) {
2    return a + b;
3  }
4
5
6
7  for(var i=0;i<5000000;i++){
8    var arg1, arg2;
9    if (i % 2 === 0) {
10     a = 1; b = 2;
11
12   } else {
13     a = 'a'; b = 'b';
14
15   }
16   f(a, b);
17 }
```

```
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);
  }

}
```

Figure 3: Example of polymorphic operation (left) and improved code (right).

```
1 var x, y;
2 var rep=300000000;
3 for(var i=0;i<rep;i++){
4   y = x | 2;
5 }
```

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

Figure 4: Example for a binary operation on `undefined` (left) and improved code (right).

operands. In the modified code, the operation at line 2 always executes on numbers, and the operation at line 5 always executes on strings. For example, the JIT compiler can optimize the monomorphic plus into a few quick integer instructions and inline these instructions at the call site of `f`. In contrast, the JIT compiler cannot optimize the original code because the types of operands change every time line 2 executes.

2.3. Binary Operation on `undefined`

Binary operations, such as `+`, `-`, `*`, `/`, `%`, `|`, and `&`, that are executed on `undefined` values (which has well-defined semantics in JavaScript), can cause a loss of performance compared to applying the same operations on non-`undefined` values.

Example. The code on the left of Figure 4 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 significantly improves the performance. The modified code on the right is 1.8% and 82.8% faster than the original code in Firefox and Chrome, respectively.

Explanation. The reason for this performance difference is that 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 the `undefined` value into a number.

2.4. 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.

Example. The code on the left of Figure 5 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

```
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 5: Example of non-contiguous arrays (left) and improved code (right).

```
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 6: Example of accessing undefined array elements.

contiguously leads to a significant performance improvement. The modified code on the right is 97.5% and 90.2% faster than the original code 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, are represented using linear storage. Sparse arrays, where keys are non-contiguous, are implemented as hash tables, and looking up elements is relatively slow. Second, using non-contiguous arrays may degrade performance because the JavaScript engine may change the representation of an array if its density changes during the execution. Third, non-contiguous arrays are JIT-unfriendly because JIT compilers speculatively specialize code under the assumption that arrays do not have holes. For example, Hackett et al. [17] describe a type inference-based JIT optimization implemented in Firefox, which marks arrays as contiguous or potentially non-contiguous (called "packed" and "unpacked" in [17]) and which generates optimized code for contiguous arrays.

2.5. Accessing Undefined Array Elements

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

Example. The code in Figure 6 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 and instead continues the `while` loop as long as the current index `ij` is smaller than the length of the array. This change results in a significant performance improvement of 73.9% and 70.2% in Firefox and Chrome, respectively.

Explanation. Accessing undefined array elements causes slowdown for reasons similar to ones discussed in Section 2.4.

```
var array = [];
for(var i=0;i<10000000;i++)
  array[i] = i/10;
array[4] = "abc";
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 7: Example of storing non-numeric values into numeric arrays.

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%

Table 1: Performance improvements on micro-benchmarks of JIT-unfriendly code patterns.

2.6. Storing Non-numeric Values in Numeric Arrays

JavaScript arrays may contain elements of different types. However, for good performance, programmers should avoid to store non-numeric values into an otherwise numeric array.

Example. The code on the left of Figure 7 creates a large array that contains only numeric values. Then, the code assigns a non-numeric value to one of the array's elements. The modified code on the right avoids storing a non-numeric value, which leads to a significant performance improvement of 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 [4] or doubles, then the JavaScript engine can represent the array as a fixed sized C-like array of integers or doubles, respectively. Operations on such arrays are faster than on arrays containing values of arbitrary types. If a numeric array in a program is updated with a non-numeric value, the JavaScript engine must change the representation of the array from a fixed-sized integer/double array to an array of non-numeric values, which is an expensive operation.

Table 1 summarizes the JIT-unfriendly code patterns and the performance improvement we measure for micro-benchmarks when avoiding these patterns.

3. Dynamic Analyses to Detect JIT-unfriendly Code Patterns

The previous section describes several code patterns that prohibit profitable JIT optimizations. In this section, we describe JITPROF, a profiling approach that detects these code patterns at runtime and reports them to the developer. Our experience with JITPROF shows that JIT-unfriendly code reported by JITPROF often causes noticeable performance bottlenecks, and that developers can optimize JavaScript code by refactoring JIT-unfriendly code into JIT-friendly code.

[4] Both the Firefox and the Chrome JavaScript engine use tagged integers [7], where 31-bit represent a signed integer and the remaining bit distinguishes an integer value from a pointer.

To detect instances of the code patterns described in the previous section, JITPROF keeps track of particular operations that happen at runtime. For example, the analysis keeps track of operations that read and write properties of an object, called put and get property operations, respectively, and of binary operations, such as arithmetic and logical operations. For each code pattern, JITPROF associates a zero-initialized counter with source code locations that may execute a JIT-unfriendly operation. We call this counter the *unfriendliness counter*. Whenever the analysis observes a JIT-unfriendly operation, it increments the unfriendliness counter of the respective operation. In addition to keeping track of runtime operations, JITPROF associates meta-information, called *shadow objects*, with runtime objects. At the end of the program's execution, JITPROF reports code locations with a non-zero unfriendliness counter to the developer. We rank reports by their unfriendliness counter, i.e., by how frequently a JIT-unfriendly operation occurs at a particular code location, and (for some patterns) by an estimate of how profitable it is to fix the problem. Similar to other profiling approaches, we expect developers to inspect only the top-ranked reports.

In the following, we describe a dynamic analysis for each of the JIT-unfriendly code patterns.

3.1. Tracking Inconsistent Object Layouts

To find performance problems caused by inconsistent object layouts (Section 2.1), JITPROF tracks the hidden class associated with each object and the number of inline cache misses that occur at code locations that perform a property get or put operation. The unfriendliness counter for this code pattern represents how often a location suffers from an inline cache miss and an estimate of how profitable it is to fix the problem.

The analysis represents the hidden class of an object as a list of the object's property names. The list is stored in the shadow value associated with the object, and it represents the order in which the object's properties are initialized. This representation of hidden classes is independent of the underlying JavaScript engine and abstracts from the implementation of hidden classes in JavaScript engines. The analysis updates the hidden class associated with each object as follows:

- Whenever an object gets created using an object literal or using a constructor, the analysis 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 list, the analysis sets the list as the shadow value of the newly created object. If there is no matching list, the analysis creates a new list of property names and associates it with the object. The list is also added to a global database of hidden classes so that the analysis can later reuse it when searching for a particular hidden class.
- Whenever a put property operation is performed on an object, the analysis checks if the property name involved in the operation is already present in the hidden class of the object. If the property name exists, the analysis does noth-

ing. Otherwise, the analysis extends the hidden class by adding the new property name to the list of property names. Then, the analysis checks if the new list of names matches any existing hidden class. If a match is found, the analysis associates the matching hidden class with the object. Otherwise, the analysis creates a new hidden class with the list of property names, associates the list with the object, and adds the list to the database of hidden classes.

Based on the information about the hidden class of each object, JITPROF tracks whether property get and put operations result in inline cache misses by maintaining for each such location the following information:

- The value *cached_hidden_class*, which points to the hidden class of the base object of the get or put property operation that was executed most recently at the code location.
- The value *cached_prop_name*, which stores the name of the most recently accessed property at the code location.
- The unfriendliness counter c_{icm}, which represents the number of inline cache misses observed at this get or put property operation.

Whenever the program performs a get or put property operation, the analysis updates the information associated with the operation's code location. If the hidden class of the operation's base object matches the *cached_hidden_class* and if the accessed property matches the *cached_prop_name*, then the analysis does nothing. This case corresponds to an inline cache hit, i.e., the code specialized by the JIT compiler for this location can be executed. If the *cached_hidden_class* or the *cached_prop_name* does not match, then the analysis increments c_{icm} and updates *cached_hidden_class* and *cached_prop_name*. 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 the slower, generic code. The analysis also keeps track of the number of occurrence of each hidden class.

At the end of the execution, JITPROF reports a ranked list of all code locations with a non-zero number c_{icm} of inline cache misses. The results are ranked by the sum of c_{icm} and a number c_{second} that estimates how profitable it is to fix a problem at a particular location. The value c_{second} is the number of occurrences of the second most frequently observed hidden class at the location. This approach is a heuristic to break ties when multiple locations have similar numbers of inline cache misses. The rationale is that the location with a larger c_{second} is likely to be more profitable to fix because making the two most frequent hidden classes consistent can potentially avoid more inline cache misses.

For the example in Figure 1, the analysis tracks the hidden classes illustrated in Figure 2, identifies various inline cache misses at line 11, and reports this line to the developer, as illustrated by the highlighted code on the left of the figure.

3.2. Tracking Polymorphic Operations

To detect performance problems caused by polymorphic operations (Section 2.2), JITPROF tracks the types of operands involved in unary and binary operations. The unfriendliness counter for this code pattern represents how often an operation at a particular code location is polymorphic. The analysis maintains the following information for each code location that performs a unary or binary operation:

- The most recently observed type $op1\_type$ of the left operand.
- The most recently observed type $op2\_type$ of the right operand (for binary operations only).
- The unfriendliness counter c_{poly}, which represents the number of times that the types of the operands have changed during the program's execution.

Whenever the program performs a binary operation, the analysis checks whether the types of the operands match the stored $op1\_type$ and $op2\_type$. If the types match, then the analysis does nothing. This case corresponds to an operation that the JIT compiler can effectively optimize through pre-computed type-specific code. If at least one of the two observed types differs from the respective stored type, then the analysis increments c_{poly}, and it updates $op1\_type$ and $op2\_type$ with the current operand types. In this case, the JavaScript engine cannot execute type-specific code but must fall back on the slower, generic implementation. The analysis performs similar checks and updates for unary operations. At the end of the execution, JITPROF reports code locations with a non-zero number c_{poly} of polymorphic operations and ranks them in the same way as described in Section 3.1.

For the example in Figure 3, the analysis warns about the polymorphic operation at line 2 because the types of its operands always differ from the previously observed types.

3.3. Tracking Binary Operations on `undefined`

To detect performance problems caused by binary operations with `undefined` operands (Section 2.3), JITPROF tracks all binary operations and maintains an unfriendliness counter c_{undef} for each code location with a binary operation. The counter represents how often the operation operates on an `undefined` operand. Whenever the program executes a binary operation, the analysis checks whether one of the operands is `undefined`. In this case, the analysis increments c_{undef}. At the end of the execution, JITPROF reports the code locations with a non-zero value of c_{undef} and ranks them by the unfriendliness counter.

For the example in Figure 4, the analysis warns about line 4 because the first operand of the operation is frequently observed to be `undefined`.

3.4. Tracking Non-contiguous Arrays

To detect performance problems caused by non-contiguous arrays (Section 2.4), JITPROF tracks writes of array elements

that make the array non-contiguous. The analysis maintains an unfriendliness counter $c_{non-cont}$ for each code location that writes into an array. The counter represents 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 the index, the analysis checks whether the index of the element to be updated 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 by leaving a hole between the existing array and the inserted element. Therefore, the analysis increments the counter $c_{non-cont}$ for the code location. At the end of the execution, JITPROF reports all code locations with a non-zero $c_{non-cont}$, ranked by the unfriendliness counter.

For the example in Figure 5, the analysis warns about line 4 because it transforms the array into a non-contiguous array every time the line is executed.

3.5. Tracking Access to Undefined Array Elements

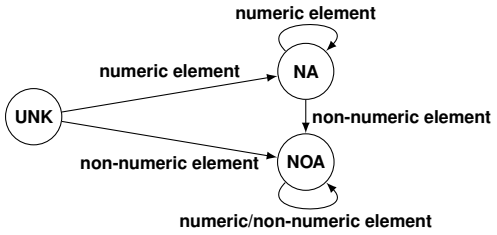
To find performance problems caused by accessing undefined array elements (Section 2.5), JITPROF tracks all operations that read array elements. The unfriendliness counter c_{uninit} for this code pattern represents how often a code location reads an undefined array element. Similar to writes into arrays, reading an array element in JavaScript is a get property operation where the base object is an array and where the property name is the index. The analysis checks for each get property operation that reads an array element from `base` whether the `index` is defined in the array by calling `base.hasOwnProperty(index)`. If this check fails, the program accesses an undefined array element, and the analysis increments the c_{uninit} counter of the code location. At the end of the execution, JITPROF reports all code locations with a non-zero number c_{uninit} of reads of undefined array elements and ranks them by the unfriendliness counter.

For the example in Figure 6, the analysis warns about line 7 because it reads an undefined array element every time the `while` loop terminates.

3.6. Tracking Non-numeric Stores into Numeric Arrays

To detect performance problems caused by transforming numeric arrays into non-numeric arrays (Section 2.6), JITPROF tracks the state of each array and operations that may change the state of arrays. The unfriendliness counter for this code pattern represents how often a particular code location transforms a numeric array into a non-numeric array.

The analysis maintains a simple state machine for each array (Figure 8). The state machine has three states: *unknown*, *numeric*, *non-numeric*. When an array gets created, the analysis initialized the state machine for this array and stores its initial state in the shadow value of the array. 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 analysis updates the state of



NA means numeric array. **NOA** means non-numeric array.
UNK means uninitialized array of unknown type.

Figure 8: State machine of an array.

an array whenever the program writes into the array through a put property operation. If the operation stores a non-numeric value in a numeric array, the state machine transitions from *numeric* to *non-numeric*. If a numeric value is stored in a numeric array, the state of the array remains unchanged. If a numeric value is stored in an *unknown* array, the state of the array transitions to *numeric*. In all other scenarios, the state of the array is changed to *non-numeric*.

The analysis maintains an unfriendliness counter $c_{non-num}$ for each code location that writes an array element. If executing an operation leads to a transition from *numeric* to *non-numeric*, then the analysis increments $c_{non-num}$ for the corresponding code location. At the end of the execution, JITPROF reports all code locations that transform numeric into non-numeric arrays, i.e., all code locations with a non-zero value of $c_{non-num}$, ranked by $c_{non-num}$.

For the example in Figure 7, the analysis warns about line 4 because it writes a non-numeric value into a numeric array.

3.7. Discussion

The dynamic analyses 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 that check for JIT-unfriendly code patterns with a few lines of code, and without requiring knowledge about the engine's implementation details. Evaluating our approach with state of the art JavaScript engines (Section 5) shows that it effectively identifies valuable optimization opportunities, suggesting that our dynamic analyses model the engine's behavior in a reasonable way.

4. Implementation

We implement JITPROF via a source-to-source transformation that adds analysis code to a given program. This approach avoids limiting JITPROF to a particular JavaScript engines. The implementation is built on top of the instrumentation and dynamic analysis framework JALANGI [33].

The framework instruments a JavaScript programs through source-to-source transformation and then, the instrumented code is executed in place of the original code. A JALANGI-instrumented JavaScript program provides two capabilities to a dynamic analysis. First, it allows the analysis to associate an arbitrary value, called *shadow value*, with any object in the program to store meta-information about the object. Second, JALANGI supports *shadow execution* on shadow values, a technique in which an analysis can update the shadow values and analysis state on each operation performed by the program. The operations considered by JALANGI are at a lower level than JavaScript statements, e.g., complex expressions are split into multiple unary and binary operations.

For each JIT-unfriendly code pattern, JITPROF contains a dynamic analysis that uses shadow values and shadow execution to identify occurrences of the pattern. For example, the analysis to detect inconsistent object layouts (Section 3.1) stores a representation of the hidden class of each object as the object's shadow value. The analysis uses shadow execution to update the hidden class whenever an object property is added or deleted. Furthermore, the analysis uses shadow execution to check at each operation that accesses a property whether to increment the unfriendliness counter. The implementation 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 that are involved in a JIT-unfriendly code pattern.

Given the JALANGI framework, the implementations of the analyses are relatively simple, making it straightforward to extend JITPROF with additional JIT-unfriendly code patterns. Our implementations of the analyses in Section 3 require between 53 and 278 lines of JavaScript code.

5. Evaluation

To evaluate the effectiveness of JITPROF, we apply it to the SunSpider [2] and Octane [1] benchmark suites. We inspect the code locations that JITPROF identifies, refactor them by replacing JIT-unfriendly code with JIT-friendly code, and measure whether these simple changes lead to a performance improvement in the Firefox and Chrome web browsers (Section 5.2). Furthermore, we compare the effectiveness of JITPROF with traditional CPU-time profiling (Section 5.3). We want to note that finding JIT compilation-related performance problems in the SunSpider and Octane benchmarks is non-trivial because these benchmarks have been used extensively to tune the performance of popular JavaScript engines.

5.1. Experimental Methodology

Table 2 lists the programs used for the evaluation, along with their number of lines of code. The table also shows the running time of the benchmark when we profile it with JITPROF and the slowdown imposed by JITPROF compared to running the benchmark without any instrumentation. Our implementation is not optimized for reducing slowdown but instead focuses on providing a JavaScript engine-independent framework that is easily extensible.

Benchmark	LOC	Time of profiling run (sec)	Profiling slowdown (×)
SunSpider-Controlflow-Recursive	25	3.61	62
SunSpider-Bitops-Bits-in-Byte	26	36.78	593
SunSpider-Bitops-Bitwise-And	31	18.05	282
SunSpider-Math-Partial-Sums	33	7.92	144
SunSpider-Bitops-Nsieve-Bits	35	23.95	380
SunSpider-Bitops-3bit-Bits-in-Byte	38	21.68	355
SunSpider-Access-Nsieve	39	19.65	333
SunSpider-Math-Spectral-Norm	51	13.92	204
SunSpider-Access-Binary-Trees	52	6.36	104
SunSpider-3d-Morph	56	14.5	219
SunSpider-String-Unpack-Code	67	4.67	55
SunSpider-Access-Fannkuch	68	62.52	893
SunSpider-String-Fasta	90	11.38	175
SunSpider-String-Validate-Input	90	0.15	2
SunSpider-Math-Cordic	101	44.7	638
SunSpider-String-Base64	136	13.31	221
SunSpider-Access-Nbody	170	28.5	438
SunSpider-Crypto-SHA1	225	8.29	133
SunSpider-String-Tagcloud	266	8.92	96
SunSpider-Crypto-MD5	288	7.47	118
SunSpider-Date-Format-Tofte	300	13.79	246
SunSpider-3d-Cube	339	50.71	8
SunSpider-Date-Format-Xparb	418	5.15	37
SunSpider-Crypto-AES	425	19.69	289
SunSpider-3d-Raytrace	443	19.9	280
SunSpider-Regexp-DNA	1,714	0.15	2
Octane-Splay	395	0.73	14
Octane-Navier-Stokes	407	168.8	2,519
Octane-Richards	537	4	70
Octane-DeltaBlue	880	7.05	113
Octane-Raytrace	904	23.46	345
Octane-Code-Load	1,527	2.9	43
Octane-Crypto	1,697	259.79	2,763
Octane-Regexp	1,765	14.42	110
Octane-Earley-Boyer	4,683	90.15	1,024
Octane-Box2d	9,537	203.13	441
Octane-Gbemu	11,106	757.01	2,482
Octane-Typescript	25,911	1,41,659	878
Octane-Pdfjs	33,062	234.9	748
Octane-Mandreel	277,375	6,201.94	3,674

Table 2: Programs used for the evaluation.

For each JIT-unfriendly code pattern, JITPROF reports a ranked list of code locations. We inspect at most the top three code locations per program and pattern, and we try to refactor the program to avoid the JIT-unfriendly code. We apply only semantics-preserving changes that affect at most a few lines of code. Each change fixes only the problem reported by JITPROF and does not apply any other optimization.

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 [9, 14, 27], 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 (explained below), giving a total of 100 scores per browser. Given these scores, we use the independent T-test (95% confidence interval) to check whether there is a statistically significant performance difference be-

Benchmark	CPR	JITPROF Rank	Ch. LOC	Avg. improvement (stat. significant)	
				Firefox	Chrome
SunSpider-Crypto-SHA1	6	1 in UAE, PO, BOU	6	3.3%	26.3%
SunSpider-String-Tagcloud	-	1 in IOL	15	-	11.7%
SunSpider-Crypto-MD5	9	1 in UAE, PO, BOU	6	-	24.6%
SunSpider-Format-Tofte	1	1 in UAE	2	-	3.4%
SunSpider-3d-Cube	5	1 in NCA	1	-	1.1%
SunSpider-Format-Xparb	1	1 in PO	2	19.7%	22.4%
SunSpider-3d-Raytrace	5	1 in NNA	4	-	2.6%
Octane-Splay	6	1 in IOL	2	3.5%	15.1%
Octane-SplayLatency	6	1 in IOL	2	-	3.8%
Octane-DeltaBlue	7	2 in IOL	6	1.4%	-
Octane-RayTrace	1	1 in IOL	18	-	12.9%
Octane-Box2D	25	2 in IOL	1	-	7.5%

CPR means CPU Profiler Rank. Ch. LOC is the number of changed LOC. IOL means inconsistent object layouts. PO means polymorphic operations. BOU means binary operation on `undefined`. NCA means non-contiguous arrays. UAE means accessing undefined array elements. NNA means storing non-numeric values in numeric arrays.

Table 3: Performance improvement achieved by avoiding JIT-unfriendly code patterns.

tween the original and the modified program. All performance differences we report are statistically significant. The experiments are performed on Mac OS X 10.9.2 using a 2.40GHz Intel Core i7-3635QM CPU machine with 8GB memory.

To assess the performance of a single benchmark execution, we rely on the measurement infrastructure that is part of the SunSpider and Octane benchmarks. SunSpider benchmarks report a performance score that is the total amount of time used to complete a fixed amount of computation, i.e., a lower score means better performance. Therefore, we compute performance improvement as $(s_{original} - s_{modified})/s_{original}$ for SunSpider, where $s_{original}$ and $s_{modified}$ mean the average score for the original and modified benchmark, respectively. Since some SunSpider benchmarks run too short to reliably measure their performance, we modify their test harness code by increasing the number of repetitions of the computation. Octane benchmarks report a performance score that is proportional to the number of repetitions performed in a fixed amount of time, i.e., a higher score means better performance. Therefore, we compute performance improvement as $(s_{modified} - s_{original})/s_{original}$ for Octane.

5.2. JIT-unfriendly Code Found by JITPROF

JITPROF detects JIT-unfriendly code that causes easy to avoid performance problems in 12 of the 40 benchmarks. Table 3 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 how many lines of code we change to avoid the problem. The last two columns of the table show the performance improvement achieved with these changes. Avoiding JIT-unfriendly code patterns leads to improvements of up to 19.7% and 26.3% in Firefox and Chrome, respectively. In the following, we discuss representative examples of performance problems and how to avoid them. The Appendix list all examples from Table 3 that we are not discussed in this section.

Inconsistent Object Layouts in Octane-Splay. JITPROF reports a code location where inconsistent object layouts occur a total of 135 times. Specifically, the layout of the objects frequently alternate between two layouts: `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 depending on the outcome of the conditional check at line 2:

```
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 }
```

To fix the problem, we swap the first two statements in the `else` branch so that the code always creates an object with layout `key|value|left|right`. This simple change results in a 3.5% and 15.1% improvement in Firefox and Chrome, respectively.

Polymorphic operations in SunSpider-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 caused by 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) {
4      ch = " ";
5    }
6    while (result.length < size){
7      result = ch + result;
8    }
9    return result;
10 }
```

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 without the statement that calls the `String` constructor, which is the solution a developer may choose in practice, leads to an even larger improvement.

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 {
3   var bin = Array();
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 function creates an empty array and then 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 6. Furthermore, the approach reports that this line repeatedly performs bitwise OR operations on the `undefined` value. Finally, the approach also reports that this operation is polymorphic because it operates both on numbers and on `undefined`.

To avoid this conglomerate of JIT-unfriendly operations, we refactor the code as follows:

```
1 function str2binl(str)
2 {
3   var len = (str.length * chrsz)>>5;
4   var bin = new Array(len);
5   for (var i = 0; i < len; i++) bin[i] = 0;
6   var mask = (1 << chrsz) - 1;
7   for (var i = 0; i < str.length * chrsz; i += chrsz)
8     bin[i>>5] |= (str.charCodeAt(i/chrsz) & mask)<<(i%32);
9   return bin;
10 }
```

The modified code avoids all three JIT-unfriendly code patterns. It 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% performance 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 5: 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 performance improvement of 1.1% in Chrome.

Non-numeric Values in Numeric Arrays in SunSpider-Raytrace. JITPROF reports that the SunSpider-Raytrace benchmark transforms a numeric array into a non-numeric array 30 times. The reason is that the program initializes an array that is supposed to represent pixels with 30 zeros:

```
1 var size = 30;
2 var pixels = new Array();
3 for (var y = 0; y < size; y++) {
4   pixels[y] = new Array();
5   for (var x = 0; x < size; x++) {
6     pixels[y][x] = 0;
7   }
8 }
```

The code initializes `pixels[y]` as a numeric array, but the program later stores 30 non-numeric values into the array that each represent a pixel (each pixel is a numeric array of length three). Since the initial zero values are never used in the program, we refactor the initialization code as follows:

```
1 var size = 30;
2 var tmp = [0,0,0];
3 var pixels = new Array();
4 for (var y = 0; y < size; y++) {
5   for (var x = 0; x < size; x++) {
6     pixels[y][x] = tmp;
7   }
8 }
```

The new code informs the JIT engine that the array should contain only array values, which avoids adapting the data representation at runtime. The change improves performance by 2.6% in Chrome.

5.3. Comparison with CPU-time Profiling

The most prevalent existing approach for finding performance bottlenecks is CPU-time profiling [15]. To compare JITPROF with CPU-time profiling, we analyze the benchmark programs in Table 3 with the Firebug Profiler.[5] 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 3 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 CPU 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 performance problems. 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 an optimization opportunity 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.

5.4. 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

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

in an unpredictable way at runtime, but we cannot easily refactor such problems. Another common pattern is JIT-unfriendly code that is not executed frequently and where eliminating the JIT-unfriendly code requires adding statements. For example, sometimes creating consistent object layouts requires adding some 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 inspecting and optimizing such code by inspecting only the top-ranked JIT-unfriendly code locations, which occur relatively often.

6. Related Work

6.1. Just-in-time Compilation

JIT compilation is widely used to improve the performance of a program while it executes [4]. Recent work includes trace-based dynamic type specialization [13], optimizing the representation of arrays based on object access profiles [30], memoizing side effect-free methods [42], identifying and removing short-lived objects [34], just-in-time value specialization [8], and studying how the effectiveness of JIT compilation depends on the order in which compilation units are (re)-compiled [11]. Hackett et al. [17] improve the performance of the Firefox JavaScript engine through combined static-dynamic type inference that enables the engine to omit unnecessary runtime checks, given that the program matches particular regularity assumptions. Ahn et al. improve Chrome's JavaScript engine by modifying the structure of hidden classes to increase the inline caching hit rate [3]. These approaches improve the performance of existing programs, whereas JITPROF pinpoints code locations that developers may refactor to improve performance on existing JavaScript engines. Even though we expect further improvements of JIT compilation in the future, we also expect that there will always remain code that cannot be compiled into efficient machine code by a given JIT compiler. Our work addresses the problem of identifying such code.

6.2. Performance Analysis and Profiling

A recent study [21] shows that performance bugs are a common problem. In the following, we discuss existing approaches to detect and diagnose such problems. St-Amour et al. [36] propose to instrument a compiler so that it creates a log of all optimization decisions and to use this log to suggest to the developer code changes that enable currently missed optimizations. In contrast to this compile time analysis implemented inside a compiler, JITPROF is a runtime analysis that is implemented without modifying the JavaScript engine.

JavaScript developers often rely on web sites that compare the execution time of particular code snippets across JavaScript engines[6] or on advice on how to write efficient JavaScript code [45]. In contrast to these generic and program-agnostic guidelines, our approach points to optimization opportunities in a given program.

Hauswirth et al. [19] propose multi-layer profiling to understand the whole-system performance of programs executed in a virtual machine. PowerScope [12] is a profiler for measuring energy consumptions at procedure and process level. CProf [24] and StatCache [6] are cache profiling systems that identify memory cache misses in frequently executed parts of a program. These approaches and our work share the idea of analyzing the interaction between a program and its execution environment. PerfDiff is a dynamic analysis to understand and localize performance differences between execution environments [46]. Instead, our work pinpoints performance problems that may exist in multiple execution environments.

There are various profilers and dynamic analyses to detect inefficient memory usage and other performance bottlenecks. Xu et al. propose approaches to find underutilized or overutilized containers [39], unnecessarily copied data [40], and objects where the cost to create the object exceeds the benefit from using it [41]. Yan et al. detect common patterns of excessive memory usage through reference propagation profiling [43]. TAEDS is a framework to record and analyze data structure evolution during the execution [37]. Marinov and O'Callahan propose an analysis to find optimization opportunities due to equal objects [26], and Xu refines this idea to detect allocation sites where similar objects are created repeatedly [38]. Toddler [29] detects loops where all iterations have similar memory access patterns and proposes to move the corresponding code out of the loop. Profiling is also used to understand the performance of interactive user interface applications [22, 31] and large-scale, parallel HPC programs [5, 35]. Pin is a runtime instrumentation framework for binaries that includes profilers and cache simulators [25]. Other approaches combine execution traces from multiple users to localize performance problems [18, 44]. All these approaches detect performance problems that are independent of the program's execution environment. Instead, JITPROF focuses on JIT-unfriendly code locations. LLVM [23] is a framework to support program analysis at compiler level. This framework is capable of gather profiling information for later analysis and optimization. Mytkowicz et al. [28] compare four Java profilers and show that non-uniform sampling may skew the profiling results. Our implementation of JITPROF avoids this problem by not sampling the execution.

7. Conclusion

This paper presents JITPROF, a profiling framework to pinpoint code locations that prohibit profitable JIT optimizations. We realize our idea for six code patterns that lead to performance bottlenecks on popular JavaScript engines and show that JITPROF finds instances of these patterns in widely used benchmark programs. Simple changes of the programs to avoid the JIT-unfriendly code patterns lead to significant performance improvements of up to 26.3%. Our work is the

[6] For example, http://jsperf.com.

foundation for an easy to use tool that pinpoints JIT-related optimization opportunities without the need to fully understand the JavaScript engine. 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.

Acknowledgment

This research is supported in part by NSF Grants CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, and gifts from Mozilla and Samsung. The authors would like to thank Luca Della Toffola for his valuable feedback.

References

[1] Octane Benchmark Suite (v1). https://developers.google.com/octane/.

[2] SunSpider Benchmark Suite. https://www.webkit.org/perf/sunspider/sunspider.html.

[3] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, , and Josep Torrellas. Improving JavaScript performance by deconstructing the type system. In *PLDI*, 2014.

[4] John Aycock. A brief history of just-in-time. pages 97–113, 2003.

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

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

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

[8] Igor Costa, Péricles Alves, Henrique Nazare Santos, and Fernando Magno Quintão Pereira. Just-in-time value specialization. In *CGO*, pages 1–11, 2013.

[9] Charlie Curtsinger and Emery D. Berger. STABILIZER: statistically sound performance evaluation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 219–228. ACM, 2013.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, pages 297–302, 1984.

[11] Yufei Ding, Mingzhou Zhou, Zhijia Zhao, Sarah Eisenstat, and Xipeng Shen. Finding the limit: examining the potential and complexity of compilation scheduling for jit-based runtime systems. In *ASPLOS*, pages 607–622, 2014.

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

[13] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478, 2009.

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, pages 57–76. ACM, 2007.

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

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

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

[18] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155. IEEE, 2012.

[19] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael 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.

[20] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 1991.

[21] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[22] Milan Jovic, Andrea Adamoli, and Matthias 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.

[23] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

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

[25] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM, 2005.

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

[27] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276, 2009.

[28] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 187–197. ACM, 2010.

[29] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571. IEEE / ACM, 2013.

[30] Rei Odaira and Toshio Nakatani. Continuous object access profiling and optimizations to overcome the memory wall and bloat. In *ASPLOS*, pages 147–158, 2012.

[31] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Conference on Operating Systems Design and Implementation (OSDI)*, pages 107–120. USENIX, 2012.

[32] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, pages 1–12. ACM, 2010.

[33] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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.

[34] Ajeet Shankar, Matthew Arnold, and Rastislav 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.

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

[36] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *OOPSLA*, pages 163–178, 2012.

[37] Xiao Xiao, Jinguo Zhou, and Charles Zhang. Tracking data structures for postmortem analysis. In *ICSE*, pages 896–899. ACM, 2011.

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

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

[40] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary 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.

[41] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

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

[43] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering, (ICSE)*, pages 134–144. IEEE, 2012.

[44] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: a device-driver case. In *ASPLOS*, pages 193–206, 2014.

[45] Nicholas C. Zakas. High performance javascript - build faster web application interfaces. 2010.

[46] Xiaotong Zhuang, Suhyun Kim, Mauricio J. Serrano, and Jong-Deok 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.

Appendix

JITPROF detects some JIT-unfriendly code locations that we do not discuss in Section 5.2 for space reasons. The Appendix describes these remaining examples.

Access to Undefined Array Elements in SunSpider-Format-Tofte. JITPROF reports that the following code accesses an undefined array element 1,000 times:

```
var ia = input.split('');
var ij=0;
while (ia[ij]) {
  ...
  ij++;
}
```

The `while` loop iterates over the elements of the array starting from index 0 and performs some operations on each element. To check if the current index is within the bounds of the array, the code checks if the element of the array at the current index is defined or not. As a result, the code accesses an undefined array elements every time it is executed. The undefined array access can be eliminated by refactoring the code as follows:

```
var ia = input.split('');
var ij=0, var len = ia.length, var sum = 0;
while (ij < len) {
  sum += ia[ij];
  ...
  ij++;
}
```

In the refactored code, the `while` loop uses the length of the array to make sure that the current index is always within the bounds of the array. For a fair comparison, we also add an additional load operation inside the loop at line 4. The modified code yields a 3.4% improvement in Chrome.

Inconsistent Object Layouts in Octane-DeltaBlue. JITPROF reports a code location where two inconsistent object layouts occur 11,700 and 2,017 times, respectively. The objects that occur at this code location frequently alternate between two layouts: `strength|v1|v2|direction` and `direction|scale|offset|strength|v1|v2`. The reason for this JIT-unfriendly code is the following code:

```
function BinaryConstraint(var1, var2, strength) {
  ...
  this.v1 = var1;
  this.v2 = var2;
  this.direction = Direction.NONE;
  ...
}

BinaryConstraint.prototype.output = function() {
  return (this.direction == Direction.FORWARD) ?
      this.v2 : this.v1;
}
...
EqualityConstraint.inheritsFrom(BinaryConstraint);
...
ScaleConstraint.inheritsFrom(BinaryConstraint);
```

Objects of type `EqualityConstraint` and objects of type `ScaleConstraint` both inherit from `BinaryConstraint`, which has a method `output` that uses the property `direction`, `v2` and `v1` of the base object. Since `EqualityConstraint` and `ScaleConstraint` have different structures, accessing the property `this.direction` inside `output` can lead to inline cache misses.

To eliminate these inline cache miss, we append the `output` method to the prototype object of `EqualityConstraint` and `ScaleConstraint`, respectively:

```
EqualityConstraint.prototype.output = function() {
  return (this.direction == Direction.FORWARD) ?
    this.v2 : this.v1;
}

ScaleConstraint.prototype.output = function() {
  return (this.direction == Direction.FORWARD) ?
    this.v2 : this.v1;
}
```

As a result, the layout of `this` is always consistent when accessing `constraint.output` inside the method. The refactored program results in a small but statistically significant performance improvement of 1.4% in Firefox.

Inconsistent Object Layouts in Octane-RayTrace. JITPROF reports that two inconsistent object layouts occur 26,298 and 25,042 times, respectively, at a particular code location. The inline cache misses happen because the layout of the objects involved in the property access frequently alternate between two layouts that share the same structure but that have different `prototype` objects. Some JavaScript engines treat hidden classes that point to different prototype objects as different hidden classes [3]. The code location reported by JITPROF is `this.initialize` at line 4 of the following code:

```
var Class = {
  create: function() {
    return function() {
      this.initialize.apply(this, arguments);
    }
  }
};
...
Flog.RayTracer.Color = Class.create();
...
Flog.RayTracer.Light = Class.create();
...
```

Objects of type `Flog.RayTracer.Color` and objects of type `Flog.RayTracer.Light` are both created in the function returned by `Class.create`. Each call of `Class.create` returns a new function instance, which has a fresh prototype object. As a result, each time the program initializes objects of type `Flog.RayTracer.Color` and objects of type `Flog.RayTracer.Light`, they refer to different prototype objects.

To eliminate these inline cache misses, we assign the function as a literal to `Flog.RayTracer.Color` and `Flog.RayTracer.Light`, respectively, so that when using their constructors, the structure of `this` is always consistent:

```
Class.create();
Flog.RayTracer.Color = return function() {
  this.initialize.apply(this, arguments);
}

Class.create();
Flog.RayTracer.Light = return function() {
  this.initialize.apply(this, arguments);
}
```

The refactored program results in a 12.9% performance improvement in Chrome.

Inconsistent Object Layouts in Octane-Box2D. JITPROF detects inconsistent object layouts that occur 447 and 363 times, respectively. The inline cache misses happen because the object layout observed at the code location frequently alternate between these two layouts: `indexA|wA|indexB|wB|w|a` and `indexA|indexB|wA|wB|w|a`. The cause for this JIT-unfriendly behavior is the following code:

```
X = x[t.m_count];
X.indexA = m.GetSupport(w.MulTMV(s.R, P.GetNegative()));
X.wA = w.MulX(s, m.GetVertex(X.indexA));
X.indexB = r.GetSupport(w.MulTMV(v.R, P));
X.wB = w.MulX(v, r.GetVertex(X.indexB));
...

var C = t[x];
C.indexA = b.indexA[x];
C.indexB = b.indexB[x];
...
C.wA = w.MulX(f, s);
C.wB = w.MulX(r, v);
```

The object `X` gets property `wA` initialized at line 3 and then gets property `indexB` initialized at line 4. While object `C` gets property `indexB` initialized before property `wA`. Later, a get property operation retrieves `indexB` from object `C` and object `X`, which leads to inline cache misses. We refactor this JIT-unfriendly code pattern by swapping line 3 and line 4. The refactored program results in a 7.5% improvement in Chrome.

Inconsistent Object Layouts in SunSpider-String-Tag-Cloud. JITPROF reports inconsistent object layouts for the property accesses `v[i]` at lines 6 and 8 in the following code:

```
function walk(k, v) {
  var i, n;
  if (v && typeof v === 'object') {
    for (i in v) {
      if (Object.prototype.hasOwnProperty.apply(v, [i])) {
        n = walk(i, v[i]);
        if (n !== undefined) {
          v[i] = n;
        }
      }
    }
  }
  return filter(k, v);
}
```

The inline cache misses happen because the get property operation `v[i]` at line 6 and the put property operation at line 8 access two different property names (`popularity` and `tag`) in different loop iterations, even though all object layouts are the same (`[tag|popularity]`). To avoid these inline cache misses, we refactor the code as follows:

```

function walk(k, v) {
  var i,n;
  if (v && typeof v === 'object') {
    for (i in v) {
      if (Object.prototype.hasOwnProperty.apply(v, [i])) {
        if(i === 'tag') {
          n = walk(i, v.tag);
          if (n !== undefined) {
            v.tag = n;
          }
        } else if (i === 'popularity') {
          n = walk(i, v.popularity);
          if (n !== undefined) {
            v.popularity = n;
          }
        } else {
          n = walk(i, v[i]);
          if (n !== undefined) {
            v[i] = n;
          }
        }
      }
    }
  }
  return filter(k, v);
}
```

In the refactored code, we first check if the property name is `tag` or `popularity`. When the property name matches, the control flow enters the corresponding branch where the property accesses are hard-coded (e.g., `v.tag` and `v.popularity`) so that none of the property get and put operations causes any inline cache miss. We add the final else branch to make sure that the refactored code preserves the semantics. This refactoring results in a 11.7% improvement in Chrome.

Multiple Problems in SunSpider-Crypto-SHA1. JITPROF detects occurrences of multiple JIT-unfriendly code patterns in SunSpider-Crypto-SHA1, which are similar to the example in SunSpider-Crypto-MD5 (Section 5.2):

```
function str2binb(str)
{
  var bin = Array();
  var mask = (1 << chrsz) - 1;
  for(var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |=
        (str.charCodeAt(i/chrsz) & mask)<<(32-chrsz-i%32);
  return bin;
}
```

After refactoring the code in a similar way as in SunSpider-Crypto-MD5, we notice a 26.3% performance improvement in Chrome. In Firefox, we observe an improvement of 3.3% .

Non-contiguous Arrays in SunSpider-Cube. JITPROF detects that the following code creates a non-contiguous array 208 times (already briefly explained in Section 5).

```
var CurN = new Array();
var i = 5;
for (; i > -1; i--)
  CurN[i] = VMulti2(MQube, Q.Normal[i]);
```

The array `CurN` is initially empty and the `for` loop at line 3 stores array elements from higher index to lower index. As a result, `CurN` is non-contiguous in the first four of five iterations. We avoid this performance problem by modifying the loop so that it modifies the array from the lower index to the higher index. As a result, the array increases contiguously:

```
var CurN = new Array();
for (var i=0; i < 6; i++)
  CurN[i] = VMulti2(MQube, Q.Normal[i]);
```

This change results in a small but statistically significant performance improvement of 1.1% in Chrome.