Uncovering JavaScript Performance Code Smells Relevant to Type Mutations

Xiao Xiao[1], Shi Han[2], Charles Zhang[1], and Dongmei Zhang[2]

1. The Hong Kong University of Science and Technology
{richardxx,charlesz}@cse.ust.hk

2. Microsoft Research
{shihan,dongmeiz}@microsoft.com

Abstract. In dynamic typing languages such as JavaScript, object types can be mutated easily such as by adding a field to an object. However, compiler optimizations rely on a fixed set of types, unintentional type mutations can invalidate the speculative code generated by the type-feedback JIT and deteriorate the quality of compiler optimizations. Since type mutations are invisible, finding and understanding the performance issues relevant to type mutations can be an overwhelming task to programmers. We develop a tool JSweeter to detect performance bugs incurred by type mutations based on the type evolution graphs extracted from program execution. We apply JSweeter to the Octane benchmark suite and identify 46 performance issues, where 19 issues are successfully fixed with the refactoring hints generated by JSweeter and the average performance gain is 5.3% (up to 23%). The result is persuasive because those issues are hidden in such well developed benchmark programs.

1 Introduction

JavaScript has become a pivotal building block for web and mobile applications. As a dynamically typed language, considerable academic and industrial effort is invested to optimize its performance. One of the important techniques that contributed to the dramatic improvement of the speed of JavaScript is the type-feedback Just-in-time (JIT) compilation adopted by almost all modern JavaScript engines. The type-feedback JIT is a speculative technique that leverages the runtime information to generate fast code and use it in future executions if types remain unchanged [15]. Therefore, unlike statically typed languages, programmers of dynamic languages such as JavaScript can significantly influence the success rate of the speculations.

If the code conforms to some coding idioms such as asm.js [1] to restrict the type generation and variation, the type-feedback speculations, along with all dynamic optimization techniques, can be very effective. The underlying reason is that JavaScript engines such as V8 employ two contradictory designs in dealing with types: The *fat* type design and the type *equality* testing for validating speculations. The spirit of fat type design is binding certain instance specific information such as pointer to the prototype to the type. Thus, the JIT optimizers can perform aggressive optimizations to generate type-specific and more efficient code. However, a fat type is also *fragile* that programmers can easily mutate it unconsciously, such as changing the prototype of a function. Therefore, the failure rate of type equality testing, which is the key component to validate speculative assumptions for JITed code, can be high.

(a)

```
 1 function Foobar() {
 2   this.abc = 1;
 3   this.test = function (n) {
 4     this.abc = n;
 5   };
 6 }
 7 Foobar.prototype.runTest =
 8 function (N) {
 9   for (var i=0; i<N; ++i) {
10     this.test(i);
11   }
12 };
13 var N = 10000000;
14 (new Foobar()).runTest(N);
15 (new Foobar()).runTest(N);
```

(b)

```
1 Foobar.prototype.test =
2 function (n) {
3   this.abc = n;
4 };
```

Fig. 1. The "Foobar" Objects created at Line 14 and Line 15 have different types due to the method binding optimization for *test* field.

Figure 1(a) extracted from V8's user group [1] illustrates a case where the two "Foobar" objects created at Line 14 and Line 15 have different types on Google's V8 JavaScript engine, even their allocation sites are literally the same. The reason is that the field assigned to a closure instance such as *test* (Line 3) is stored in the type descriptor rather than in the object instance. This is called *method binding*, because V8 recognizes that a field referring to a closure rarely changes [4]. When calling `Foobar` again at Line 15, *test* is assigned to a different closure instance and V8 cancels method binding for *test*. Therefore, the type of the first "Foobar" object is unequal to the type of the second "Foobar" object. As a side effect, the `runTest` function optimized against the first "Foobar" object will be invalidated when it operates on the second "Foobar" object. A quick solution is moving the field *test* to the prototype of `FooBar` as shown in Figure 1(b). This simple change gains 10× speedup.

In this paper, we present a technique that can automatically recognize the performance code smells relevant to unintentional type mutations and generate a sketched execution path for programmers to understand the smell. Moreover, refactoring hints for programmers to eliminate the type mutations are also generated. For our goal, conventional profiling techniques offer insufficient help. One could use timing functions to find the expensive code fragments. However, this gives programmers a very coarse view of performance symptoms, which cannot be used to distinguish the performance issues incurred by type mutations from other causes. Since type is implicitly represented by the JavaScript engine, programmers need to link the clues from the engine logs of internal events, the JITed code, and the source code, to understand how types are generated and evolved. This bug hunting process is overwhelming for application programmers. Moreover, the engine logs vary from one engine to another. These factors make

[1] `https://groups.google.com/forum/#!topic/v8-users/0fc_SmwDCUM`

application programmers inhibitive to understand how type mutations impact performance.

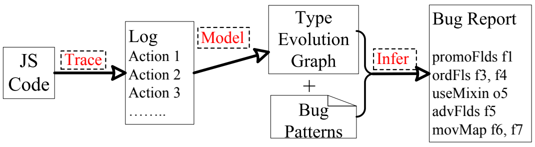

Fig. 2. Workflow of our algorithm.

We employ program execution information other than timing results to diagnose performance issues. The workflow of our technique is summarized in Figure 2. First, we trace the operations that could change the types of objects. Tracing is performed at the engine side, where changes are not required for the traced code. In the second step, we build the *type evolution graphs* (TEG), one for all the objects that are created by the same constructor. In this way, precise type evolution track for each object may be lost, but we gain the knowledge of how the objects that could have the same usage evolve to different types. Third, for each type equality testing failure, we study the path on TEG between the type expected by the testing and the type being tested, and match it to one of our predefined code patterns drawn from empirical study. If the pattern matching succeeded, we generate a refactoring suggestion.

We implement our algorithm in a tool JSweeter and apply it to the Octane benchmark suite. Our tool reports 46 performance issues relevant to type mutations. By successfully fixing 19 issues, we improve the benchmark score by up to 23%. Since the programs in Octane are all well tuned, finding performance bugs for these programs is challenging and our results are worth mentioning. In summary, our contributions are:

1. We carefully examine V8 and Firefox bug repositories and identify five common ways to cause performance issues by type mutations. Meanwhile, we identify six types of code smells that often mutate types unintentionally and conclude seven refactoring approaches to eliminate these code smells.

2. We develop an algorithm to detect the performance issues incurred by type mutations based on type evolution graph. Our approach also generates actionable refactoring suggestions by matching execution patterns to six performance issues.

3. We implement a tool JSweeter and apply it to the benchmark suite Octane. We find 46 performance bugs and 19 of the 46 issues are successfully fixed. The average speedup is 5.3% and one has significant 23% speedup.

2 Types in Type-feedback JavaScript Engine

2.1 Type Collection

Due to lack of types, JavaScript programs cannot be compiled to fully optimized binary code ahead of time. *Type-feedback* is a profiling technique that dynamically collects type information for variables [16]. The type information is fed to the JIT compiler for generating efficient speculative code. Type-feedback JITs are pervasively used by all modern browsers such as Firefox and Chrome.

(a)

```
1 function test(a, b)
2 {
3   c = a + b;
4   return c;
5 }
6 test("foo", "bar");
7 test(1, 2);
```

(b)

```
function test(a, b)
{
  if (is_str(a) && is_str(b))
    c = strcat(a, b);
  else
    c = runtime_plus(a, b);
  return c;
}
```

Fig. 3. Inline cache example. **is_str(s)** tests if s is a string. **strcat** concatenates two strings. **runtime_plus** is a runtime function to interpret the "+" operator.

The first step for type-feedback optimizations is type collection. Types are needed for interpreting the operators that have multiple semantics. For instance, the "+" operator could be applied to both numbers and strings. *Inline cache* (IC) is an effective way to collect the types and speedup the execution of the operators whose semantics depend on the types of their arguments. IC dynamically weaves the fast paths for observed types into the binary code [14]. An example is in Figure 3 (a), after the first call to `test` is executed (Line 6), a fast path for processing string is embedded into the code. We show a proof-of-concept implementation of IC in Figure 3(b), where the *if* statement is called *type guard*. When we call `test` again with string arguments, the fast path will be taken. If "a" or "b" are integers next time, such as in Line 7 of 3(a), the *else* branch is taken and the slower runtime function **runtime_plus** is called. After processing the integer arguments, a fast path for the integer type is also built, resulting in a *polymorphic IC* (PIC). Types are collected in the way of continuously patching the ICs and a JavaScript engine often provides sufficient warm-up time for type collection.

2.2 Type Mutations

Inside JavaScript engine, every piece of memory, such as an object, array, string, and closure, is associated to a *type descriptor* (TD), which is also known as *hidden class* in V8, *shape* in IonMonkey, and *structure* in JavaScriptCore. A type descriptor records certain information for correctly inferring the code behaviors such as field access. For example, a type for an object usually contains fields descriptors that describe the value type (*e.g.* integer or double) of each field and fields layout that records the offset of each field.

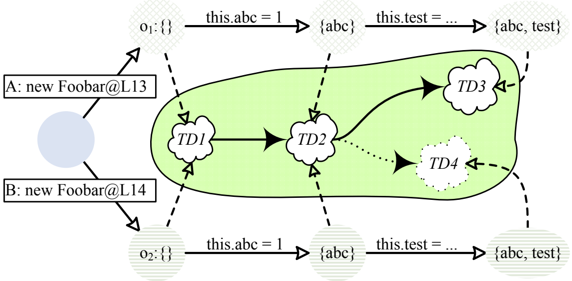

Fig. 4. Type evolution graph. Dashed arrow points to the type descriptor of corresponding object. Shadowed area is the type evolution graph and the type mutation that generates TD4 is highlighted.

Type descriptor should be *immutable* to guarantee the deterministic behavior for the code operated on. Therefore, a *type mutation* operation that changes any information in the descriptor, such as adding a field to an object, derives a new type descriptor. The set of type mutations from the same source type form a *type evolution graph*. Figure 4 shows a type evolution graph for our running example (Figure 1), where the flows with labels A and B illustrate the "Foobar" objects o_1 and o_2, created at Lines 14 and 15, respectively. From the figure, we observe that o_1 and o_2 share the first type mutation TD1 → TD2, since the statement "this.abc = 1" has the same effect on the type mutations in both executions. Later on, due to the binding of different closure instances to the same field "test", o_1 and o_2 are evolved to different types TD3 and TD4.

2.3 Why Type Mutations Impair Performance

Type mutations create new types. A large volume of types render the JavaScript engine very difficult to generate a unique piece of code that works optimally on all types. As such, programs are falling back to run with conservative runtime strategies, which are summarized as follows.

Trigger Deoptimization. Unnecessary deoptimization is a major source of performance degradation. If a hot function cannot constantly work with optimized code, its performance can be orders of magnitude worse. Moreover, frequent type changes can result in optimization-deoptimization churn and finally disable the optimization opportunity for the type unstable functions.

Trigger IC Fallback. Every IC has limited slots for building fast paths, hence saturating an IC forces some types (perhaps the frequently visited types) to be permanently handled by runtime functions.

Reduce Optimization Strength. PICs are obstacles for JIT optimizers to generate high quality code. For example, function inlining is precluded, which is a very useful optimization to enlarge the scope of intra-procedural analysis and optimizations to cross function boundary. PICs also prevent common subexpression elimination (CSE) and loop invariant code motion (LICM) to eliminate redundant type guards.

Enter Dictionary Mode. Object and array are often used as a dictionary. JavaScript engines adaptively change the backing storage of object and array to hash table in order to optimize the dictionary usage scenario. However, dictionary is manipulated by runtime functions instead of ICs, thus the fields read, write, and iteration operations slowdown significantly.

Increase GC pressure. Frequently creating and dropping small objects will increase the garbage collection (GC) frequency. High GC pressure can significantly slowdown the execution of program and increase the latency of each GC invocation, which deteriorates the user experience of interactive programs.

3 Type Mutation Code Patterns In Practice

In this section, we present our findings of learning real performance bugs from V8 and Firefox bug repositories incurred by type mutations, denoted as V8 and FF respectively. The results are summarized in Table 1. Each row contains a buggy code pattern and several representative real bug cases labeled as FF *ID*

ID		Trigger Deoptimization	Trigger IC Fallback	Reduce Opt. Strength	Enter Dictionary Mode	Increase GC Pressure	
1	Always Use New Closure	✓	✓	✓		✓	V8 2206, 2673 FF 631911, 642001
2	Inconsistent Field Ordering	✓	✓	✓			FF 813425
3	Partially Initialized objects	✓	✓	✓			FF 900849
4	Over-filled Object & Sparse Array	✓	✓		✓		V8 2734, 3313, 2192
5	Prototype Mutation	✓		✓			FF 947048, 1041126
6	Integer Overflow	✓		✓		✓	V8 2306, 2617

Table 1. Bug patterns that induce type mutations and incur performance issues.

and V8 *ID*, where ID is the bug number in corresponding repository. We identify six code patterns that mutate types and incur performance problems. For each code pattern, we also give one or more refactoring approaches from Table 2 to avoid the performance issues. These refactoring approaches are concluded from the discussion by the programmers in the bug repository.

1. Frequent Closure Creation. Similar to our running example (Figure 1), real code often creates a new closure instance before calling that function, in order to achieve better code encapsulation. However, these closure instances could result in PICs for call-sites that impair the IC efficiency and preclude inlining (V8 2206), confuse JIT and miss code optimizations opportunities (V8 2673), and increase the pressure of GC (FF 631911).

Refactoring. We can promote the fields that hold closure instances to their prototypes to avoid frequent closure creation, such as we did in Figure 1. We call this refactoring **promFlds**. If too many fields should be promoted, it is better to use the mixin design pattern to construct objects [22]. We call this way **useMixin** refactoring.

2. Inconsistent Field Ordering. JavaScript programs often have different paths to construct an object (*e.g.* by taking different *if-else* branches), and these paths add fields in different orders. For example, FF 813425 reports a real case in pdf.js: A loop randomly adds fields to the objects created from the same place, and thus, makes a hot function recompile for 11 times.

Refactoring. Guaranteeing the fields that are added in the same order can avoid generating type inconsistent objects. We name this refactoring **ordFlds**). The second suggestion is called **movMap**: Use a specialized ES6 Map [2] if an object is intended to be used as a map.

3. Partially Initialized Objects. It is common that fields are gradually added to an object during its lifetime. If the object is frequently used before fully constructed, every time the object transitioning to a new type always deoptimizes the code generated by the previous types. A dual pattern is that code is optimized against a fully constructed object. However, a partially initialized object is occasionally used and it deoptimizes the code.

Approach Abbr.	Interpretation
promFlds(f_1, ..., f_n)	Move the fields f_1, ..., f_n to its prototype
useMixin(o)	Apply mixin pattern to construct object o
ordFlds(f_1, ..., f_n)	Add the fields f_1, ..., f_n in a fixed order
movMap(f_1, ..., f_n)	Move the fields f_1, ..., f_n to an ES6 map
advFlds(f_1, ..., f_n)	Add the fields f_1, ..., f_n before use
initAry(a)	Initialize the array a before use
factorOut($srcL$)	Factor out the code around the code at $srcL$

Table 2. Refactoring approaches and the short descriptions.

Refactoring. A good practice is fully constructing an object before using it, such as adding all the fields in the constructor. We call this refactoring **advFlds**. If a derived object would like to shadow certain fields in the prototype, try to override the shadowed fields as early as possible.

4. Fat Object and Sparse Array. Adding too many fields to an object can change its backing storage to dictionary, especially adding fields via the keyed expression "p[f]" gives stronger hints than the named form "p.f" to enable the dictionary mode (*e.g.* V8 2734). If the dictionary mode is unintended, the subsequent access to the object can slowdown significantly (*e.g.* V8 3313). For arrays, the code such as "a=[]; a[x]=1;" creates a sparse array with a hole $[0, x)$. If the hole is large enough, the array is also changed to a dictionary (*e.g.* V8 2192). Moreover, accessing to a hole element returns a `undefined` value and it can invalidate ICs for operations such as "+" [7].

Refactoring. We can apply **movMap** to eliminate a fat object if most of the fields are added outside constructors. The sparse arrays can be eliminated by initializing the arrays (**initAry**). If writing to an element beyond the current array boundary is needed, try to allocate a large array and initialize it.

5. Prototype Mutation. Prototype of an object can be replaced at runtime. This behavior is popular in web libraries such as JQuery and Zepto. However, changing prototype can disable many JIT optimizations, such as the optimization for `instanceof` operator and inlining the methods in the prototype (*i.e.* FF 1041126). A more thorough discussion on this issue can be found in the bug report FF 642500.

Refactoring. Applying the mixin design pattern (**useMixin**) to construct objects is the best practice if the purpose of changing prototype is to inherit functions from different objects.

6. Integer Overflow. JavaScript only supports `double` data type, but modern JavaScript engines optimize the computations that only involve integer values. When a value exceeds integer range, a much expensive double representation such as boxed double used by V8 [10] is enabled (*i.e.* V8 2306). Moreover, if an array element overflows, the data for all the array elements will be lifted to a more general representations, as described by Bolz *et.al.* [5].

Event Name	Arguments	Interpretation
Object Events		
NewObject	***ctxt***, ***srcL***, ***obj***, ***t***	Create an object ***obj*** at line ***srcL*** under calling context ***ctxt*** with initial type ***t***
NewArray	***ctxt***, ***srcL***, ***ary***, ***t***	Create an array ***ary*** at line ***srcL*** under calling context ***ctxt*** with initial type ***t***
ChgProto	***ctxt***, ***srcL***, ***obj***, ***newProto***, ***t***	Set the prototype of ***obj*** to ***newProto*** at line ***srcL*** under calling context ***ctxt*** and change type to ***t***
NewField	***ctxt***, ***srcL***, ***obj***, ***f***, ***v***, ***md***, ***t***	Insert field ***f*** to object ***obj*** with value ***v*** at line ***srcL*** under calling context ***ctxt*** ***md***=0: ***f*** is added via ***obj***.***f*** ***md***=1: ***f*** is added via ***obj***["***f***"]
DelField	***ctxt***, ***srcL***, ***obj***, ***f***, ***t***	Delete field ***f*** of object ***obj*** at line ***srcL*** under calling context ***ctxt*** and change type to ***t***
UptField	***ctxt***, ***srcL***, ***obj***, ***f***, ***v***, ***t***	Assign value ***v*** to field ***f*** of object ***obj*** at line ***srcL*** under calling context ***ctxt*** and change type to ***t***
AryWrite	***ctxt***, ***srcL***, ***ary***, ***inx***, ***t***	Writing to array ***ary*** index ***inx*** at line ***srcL*** under calling context ***ctxt*** and change type to ***t***
RepLift	***ctxt***, ***srcL***, ***obj***, ***t***	The representation of the elements or properties in ***obj*** is lifted by executing an operation at line ***srcL*** under calling context ***ctxt*** [5]. The new representation has type ***t***
Function Events		
DeoptCode	***func***, ***obj***, ***t*** ***id***, $T_1, \ldots, T_K$	The function ***func*** deoptimized at an IC ***id*** because the type ***t*** of object ***obj*** is not previously collected by the IC ***id***. The expected types for the IC are $T_1, \ldots, T_K$.

Table 3. Definitions of the events in the operational log.

Refactoring. If overflow will eventually happen, the best solution is isolating the code that are tainted by the overflowed values to a new function, as suggested by McCutchan [19]. We call this refactoring approach **factorOut**.

4 Finding Unintentional Type Mutations

We adopt a three-step approach based on program execution information to detect the unintended type mutations and infer the refactoring suggestions. First, we capture the type mutations by runtime monitoring and construct type evolution graph. Second, we identify the unintentional type mutations by analyzing the types that incur deoptimizations. Third, we infer the bug pattern of each unintentional type mutation by analyzing the relevant part of the type evolution graph. The refactoring suggestions are naturally derived from the guidelines for refactoring the bug patterns in Section 3. More details of these steps are explained in the following sections.

4.1 Modeling Type Evolutions

We instrument the JavaScript engine to collect the *operational log*, which contains the type update operations for objects and deoptimization information. Table 3 defines all the events recorded in the log. Every object event contains the calling context information (***ctxt***) and the source code location (***srcL***) to precisely locate the event triggering code. If the value $\boldsymbol{v}$ recorded in the events *NewField* and *UptField* is a closure instance, we replace $\boldsymbol{v}$ with the unique ID of the definition place of the closure. The most important event is *DeoptCode*, which contains the types $(T_1, \ldots, T_K)$ collected at the deoptimized IC (*id*) and the object (*obj*) that causes the deoptimization.

With the operational log, we build the *type evolution graphs*, one for each *allocation source*, which is defined as follows:

Definition 1 *The allocation source AS_o for object o is:*
- *o = new ctor(...): AS_o is the constructor "ctor".*
- *o = {} or o = []: AS_o is the global unique ID that represents this object literal {} or [].*

We aggregate the objects by allocation source because *objects created by the same constructor or from the same literal likely to have the same usage scenarios, and refactoring can be easily performed at the constructor level.* In the rest of this paper, we call the objects created at the same allocation source *sibling* objects. The type evolution graph ψ for an object allocation source is defined as follows:

Definition 2 *A type evolution graph (TEG) ψ is a 6-tuple $(\Omega, S, \theta, \Sigma, \delta, q_0)$:*
- *Ω is a finite set of types.*
- *S is a finite set of states.*
- *θ: $S \to \Omega$ is an injective mapping from a state $s \in S$ to a type $t \in \Omega$. We name the reversed mapping as θ^{-1}.*
- *Σ is a finite set of events.*
- *δ: $S \times \Sigma \to S$ is a type transition function that describes a type update operation.*
- *q_0: the initial state.*

The set of type evolution graphs are collectively represented by Γ. Since the mapping between S and Ω is injective, we abuse the terms type and state in the rest of the paper.

We scan the operational log to generate the type evolution graphs. For every event in the log, we process it with Algorithm 1. The main idea of Algorithm 1 is first calling `GetTEG` to find or build the evolution graph for corresponding object. Then, it creates a state transition to reflect the type change. Other sub-procedures appeared in Algorithm 1 are explained in below:

1. `GetTEG(`$\boldsymbol{o}$`, `***newTy***`)`: Obtain the TEG for the object $\boldsymbol{o}$. If $\boldsymbol{o}$ is the first object for its allocation source, build a new TEG with initial type ***newTy***.

2. `FindState(`ψ`, o)`: Locate the state in the evolution graph ψ that contains the type of the object o at the moment.

3. `AddTransition(`s_1`, t, E)`: Create a labeled transition $s_1 \xrightarrow{E} s_2$ to reflect the type change, where s_2 is the state for type t.

The type evolution graphs created by Algorithm 1 for our running example is similar to that in Figure 4. The structure of type evolution graph is a *directed*

Algorithm 1: UpdateTEG

```
   Input: E = An event in the operational log
1  switch $E.type$ do
2      case $Object\ Event$:
3          $obj = E.obj$;
4          $newTy = E.t$;
           /* 1. Find or build an TEG */
5          $teg$ = GetTEG($obj$, $newTy$)
           /* 2: Build type transition */
6          $s$ = FindState($teg$, $obj$);
7          AddTransition($s$, $newTy$, $E$);
8          if $newTy == Dictionary$ then
9              $hint$ = ProcessDictObj ($o$);
10             EmitSuggests (hint);
11         end
12     end
13     case $Function\ Event$:
14         CheckDeopt($E.obj$, $[T_1, T_2, \ldots, T_K]$);
15     end
16 endsw
```

Algorithm 2: ProcessDictObj

```
   Input: $o$: The object in dictionary mode
1  if $o$ is object then
2      if $o$ has more than $K_f$ fields then
3          if CountKeyedAddFlds ($o$) $> 0$
           then
4              SetWatch ($o$)
5      end
6  else if $o$ is array then
7      $evt$ = last event for $o$;
8      if $evt == AryWrite$ And $evt.inx >$
       $o.length$ then
9          if $evt.inx \leq 1{,}000{,}000$ then
10             return initAry($o$);
11         end
12     end
13 end
```

acyclic graph (DAG), because type evolution cannot go back to an old type. However, two different types can evolve to the same type. For example, all dictionary mode objects have the same type.

If an object is changed to dictionary (Line 8), we infer whether or not the dictionary backing storage is intentional with Algorithm 2. First, we only consider an object with more than K_f (*e.g.* $K_f = 15$) fields as a candidate of fat object. Second, if there is at least one field of o added through the keyed expression such as "p[f]" (obtained by `CountKeyedAddFlds`), we mark the object o by `SetWatch`. The reason is adding fields to an object through keyed expression "p[f]" strongly implies that the field name "f" is only known at runtime. Hence, o is very possibly to be a dictionary. However, this heuristic alone is not enough, we need more evidence and hence we make decision in Algorithm 5. If the object is an array, we emit an **initAry** suggestion if the last event is an out-of-bound access and the array size is small enough. Access out-of-bound on a large array is very likely to use the array as a dictionary.

4.2 Checking Type Homogeneity

We define that type t_1 is *homogeneous* to type t_2 if they belong to the same TEG ψ. We use homogeneity to identify the types that are evolved from the same allocation source. In term of graph reachability, two types can be homogeneous in three ways. Suppose R_ψ is the reachability relation on ψ, where $R_\psi(x, y)$ means there is a path $x \rightsquigarrow y$ on ψ. Two types t_1 and t_2 are homogeneous iff:

- $R_\psi(t_1, t_2)$, or
- $R_\psi(t_2, t_1)$, or
- $\exists t_3 \in \Omega_\psi$, $R_\psi(t_3, t_1)$ and $R_\psi(t_3, t_2)$.

Algorithm 3: CheckDeopt

```
    Input: $o$, $t_o$: object $o$ with the type $t_o$
    Input: $id$, $[T_1, T_2, \ldots, T_K]$: $T_1, \cdots, T_k$ are the types collected by the IC $id$
 1  $nhomo$ = K;
 2  $Q = \emptyset$;
 3  $s_o$ = MapToState ($\psi_o$, $t_o$);
 4  for $t_c \in [T_1, T_2, \ldots, T_K]$ do
 5      $s_c$ = MapToState ($\psi_o$, $t_c$);
 6      if $s_c$ is non-exist then                          // $t_o$ is not homogeneous to $t_c$
 7          $nhomo = nhomo - 1$;
 8          continue;
 9      end
        // Get the path $P$ and the path distance between $t_o$ and $t_c$ on $\psi_o$
10      $P$, $d$ = ComputePath($t_o$, $t_c$);
11      if $d > 0$ then                                              // $R_\psi(t_o, t_c)$
12          $Q = Q \cup$ HandleFutureType($d$, $P$);
13      else if $d < 0$ then                                         // $R_\psi(t_c, t_o)$
14          $Q = Q \cup$ HandlePastType($-d$, $P$);
15      else                                     // $R_\psi(t_s, t_o)$ and $R_\psi(t_s, t_c)$
16          $t_s$ = FindLCA($t_o$, $t_c$);
17          $P_o = P_c = \emptyset$;
18          SplitPaths ($t_s$, $P$, $P_o$, $P_c$);
19          $Q = Q \cup$ HandleSplitType($t_s$, $P_o$, $P_c$);
20      end
21  end
22  if $CountDeoptSite(id) > K_s$ then $Q = Q \cup$ factorOut($id$);
23  if $\frac{nhomo}{K} \geq \pi_h$ then EmitSuggests($Q$);
```

We implement the homogeneity testing in Algorithm 3. The high level workflow, excluding the details in the *if ... else* block from Line 11 to Line 21, is checking the relationship between type t_o and type t_c, where t_o is the type of the object o at the time of causing deoptimization and $t_c \in [T_1, T_2, \ldots, T_K]$ is a type needed by the IC at the deoptimization site. To decide how t_o is homogeneous to t_c, we use two auxiliary procedures:

1. MapToState: It is exactly the θ^{-1} function (recall Definition 2). If the state for t_c is non-exist, t_c and t_o is not homogeneous.

2. ComputePath: It computes the shortest path P between t_o and t_c on ψ_o. If multiple paths exist, choose arbitrary one. The choice of the path does not matter, because after the refactoring, we can run the analysis again to study another path. The second return value d is the length of P. The sign of d encodes the path direction: $d > 0$ indicates $R_\psi(t_o, t_c)$. $d < 0$ represents $R_\psi(t_c, t_o)$. $d = 0$ means t_c and t_o are reachable by an intermediate node t_s.

We record how many types cached at the IC are homogeneous to t_o in the variable *nhomo*. If the ratio $\frac{nhomo}{K}$ exceeds the threshold π_h, we decide t_o as an unintentional type and output the refactoring suggestions.

Algorithm 5: HandlePastType

```
Input: $P$, $d$: The shortest path $P$ for $t_c \rightsquigarrow t_o$ with distance $d$
$E = \emptyset$, $R = \emptyset$;
$hasOtherEvents$ = false;
if IsDictMode $(o)$ And IsWatched $(o)$ then
    $R = R \cup$ movMap($o$)
end
foreach $evt \in P$ do
    if $evt$ != $NewField$ then
        if $evt$ == $ChgProto$ then $R = R \cup$ useMixin($o$);
        else if $evt$ == $RepLift$ And NumFields (o) $> K_i$ then
            $R = R \cup$ factorOut($srcL$);
        end
        $hasOtherEvents$ = true;
    end
    $E = E \cup evt$;
end
if $d \leq K_d$ And $hasOtherEvents$ == false then $R = R \cup$ advFlds($E$);
return $R$;
```

4.3 Inferring the Reason of Deoptimization

The *if . . . else* branch from Line 11 to Line 21 in Algorithm 3 infers bug patterns from the path between t_o and t_c on the type evolution graph. Since the path only has three cases, our inference algorithm works in three ways:

Algorithm 4: HandleFutureType

```
Input: $P$, $d$: The shortest path $P$ for
       $t_o \rightsquigarrow t_c$ with distance $d$
$E = \emptyset$;
if $d \leq K_d$ then
    foreach $evt \in P$ do
        if $evt$ != $NewField$ then
            return
        $E = E \cup evt$;
    end
end
return advFlds($E$);
```

1. `HandleFutureType(`d`, `P`)`: It handles the case where t_c might be a type for object o in future. This case is probably that o is used before fully constructed compared to its sibling objects, which is an instance of *partially initialized objects* bug (pattern 3). If $d \leq K_d$ and all the events between t_c and t_o are *NewField*, we emit an **advFlds** suggestion. We typically choose a small value for K_d (*e.g.* $K_d = 3$), because shorter path is more likely to be exceptional. All events should be *NewField* because advancing the *UptField* and *DelField* events are unsafe.

2. `HandlePastType(`d`, `P`)`: This situation is object o or its sibling objects have type t_c in the past. We examine the evolution path $t_c \rightsquigarrow t_o$ to confirm the bug pattern for o. First, if the backing storage of o is dictionary and o is watched at Line 4 of Algorithm 2, we deem the object o has refactoring value and emit a **movMap** suggestion. Second, if there is a *ChgProto* event on the path, we emit a **useMixin** suggestion. Third, if integer overflows and changes the value representation (*e.g. int* → *double*), we emit a **factorOut** suggestion if the object has more than K_i fields or array elements. The objects with more fields are potentially accessed in more places and thus, incur more IC failures and

Algorithm 6: HandleSplitType

```
Input: $t_s$, $P_o$, $P_c$: The paths $P_o$: $t_s \rightsquigarrow t_o$ and $P_c$: $t_s \rightsquigarrow t_c$
// $fpos$, $cls$: Mapping from field name to path position and to closure ID
$fpos = cls = \emptyset$;
for $i \leftarrow$ to $len(P_o)$ do
    $evt = P_o[i]$ ;                                   // Get $i^{th}$ event on the path $P_o$
    if $evt == NewField$ Or $evt == UptField$ then
        $v =$ evt.v;
        if $v$ is closure then $cls$[evt.f] $= v$;
        if $evt == NewField$ then $fpos$[evt.f] $= i$;
    end
end
$proF = ordF = \emptyset$;
for $i \leftarrow$ to $len(P_c)$ do
    $evt = P_o[i]$ ;                                   // Get $i^{th}$ event on the path $P_c$
    if $evt == NewField$ Or $evt == UptField$ then
        $f =$ evt.f;   $v =$ evt.v;
        if $v$ is closure instance And $cls[f] == v$ then $proF = proF \cup f$;
        if $evt == NewField$ And $fpos[f]$ != $i$ then $ordF = ordF \cup f$;
    end
end
$R = \emptyset$;
if $|proF| > K_p$ then $R = R \cup$ useMixin($o$);
else if $|proF| > 0$ then $R = R \cup$ promFlds($proF$);
if $|ordF| > 0$ then $R = R \cup$ ordFlds($ordF$);
return $R$;
```

create higher performance impact. Finally, same to Algorithm 4, if all the events between t_c and t_o are *NewField* and $d \leq K_d$, it could be a *partially initialized objects* case and we emit a **advFlds** suggestion.

3. `HandleSplitType(`t_s`,` P_o`,` P_c`)`: This case states that t_c and t_o deviate to different evolution paths at the state t_s, which is the lowest common ancestor (LCA) for t_c and t_o, computed by `FindLCA`. We first bisect the path into $t_s \rightsquigarrow t_o$ and $t_s \rightsquigarrow t_c$ two segments with `SplitPaths` in Algorithm 6. Then, we scan the two paths and fill and collect the fields that are assigned to different closure instances and the fields that are added in different order in the two paths. The scan results are stored in *proF* and *ordF*. We emit a refactoring suggestion **useMixin** if *proF* has more than K_p (*e.g.* $K_p = 7$) results, in which case using mixin pattern is better than promoting many fields to the prototype. Otherwise, if *proF* and *ordF* are non-empty, we give the **promFlds** and **ordFlds** refactoring suggestions.

We also count the number of deoptimizations incurred by each deoptimization site via `CountDeoptSite` at Line 22 of Algorithm 3. The counting result tells us which IC is less stable than others. In case the deoptimization is hard to be eliminated, we emit a **factorOut** refactoring suggestion, since factoring the code around the problematic IC site to a new function can limit the performance impact to a smaller scope. This is especially useful for performance problems happened inside a hot loop [19].

5 Evaluation

We implement our algorithm in a tool JSweeter. Operational log collection is performed on a modified version of V8. We apply JSweeter to Octane benchmark suite Version 2. The reason to choose Octane is twofold. First, we only modify V8, which is incapable to execute the JavaScript programs requiring external facilities, such as DOM and AJAX. We did not manage to modify a full functional JavaScript execution tool such as Chrome due to the excessive hacking efforts. Second, compared to other popular JavaScript benchmark suites such as Kraken and SunSpider, Octane has much larger programs modified from real world applications (up to 33,000 LOC for pdfjs) that can prove the effectiveness of our proposed algorithm for real sized programs. Our experiments are conducted on a machine running 32-bit Ubuntu 12.04 with an Intel Core2 3.0GHz CPU and 4GB RAM.

5.1 Overall Results discussion

	crypto	splay	box2d	gbemu	typescript	pdfjs
#Total Issues	4	1	8	12	18	3
#Fixed Issues	3	1	3	5	5	2
Score Before fix	18840	9362	20347	38748	19590	13858
Score After fix	19495	11480	21125	40237	20394	14330
Speedup	3.5%	23.0%	3.8%	3.8%	4.1%	3.4%

Table 4. The benchmark scores before and after fixing the performance issues.

We empirically choose the parameters $\pi_h = 0.5$, $K_d = 3$, $K_p = 7$, $K_i = 25$ and run JSweeter. Our findings are given in Table 4. The subjects that only have marginal improvements, such as zlib.js, datablue.js, and *etc.*, are omitted. We totally report 46 performance issues, which is surprising since these programs are well tuned. We successfully fix 19 of these issues that are simple enough to fix in one hour following the refactoring suggestions. The remaining 27 issues cannot be fixed in two reasons:

1. We are unable to understand 21 issues. The major reason is JSweeter only records one level calling context information for type update events, which is insufficient to guide us to trace back to the source of bug introducing place, especially we are not the authors of these programs. The benefit is that our approach incurs low overhead for collecting execution information. A tool such as that described by Feldthaus *et al.* [6] would be helpful and we will explore it in future.

2. We are unable to apply 6 refactoring suggestions (false positives). This is because our algorithm is a pure dynamic analysis without considering the static program semantics. For example, an **advFlds** suggests adding a field in the constructor, but the name of that field is extracted from user input and it is unable to add such fields in advance. Even a field *f* whose name is statically known, blindly adding *f* in the constructor can suppress the field existence testing such as *if (p.f == undefined)* and possibly change the program behaviour. Moreover, the initial value of the field is sometimes hard to determine. In future work, we

will consider using static information to weed out infeasible fixes and guide the refactoring.

We measure the benefits of refactoring the programs by Octane score, which is inversely proportional to execution time and the larger the better. The scores are obtained by a fresh checkout of V8 (version 3.29.42). For each program, we run it for five times and obtain its average score. All of the refactored programs gain higher scores, where most of the programs only have 3% – 4% speedup and one case splay.js is 23% faster. The results are indicative and cost benefit, since the JavaScript engine developers often tried hard and achieve the similar results. It is valuable to mention that JSweeter also found the bug reported in FF 813425 bug case. This bug is one of our two findings of *inconsistent field ordering* bugs in pdfjs. By adding the fields before use, we obtain 2.2% speedup for this single modification, very close to the 2.7% speedup achieved by the pdfjs developers.

5.2 Case Studies for Octane

We select five issues from three programs for case study. These cases are selected because each of them represents a different bug pattern. Also, these issues are difficult to be observed by programmers, since the bug introducing place and the symptom place are spatially far.

```
 1 SplayTree.prototype.insert =
 2 function(key, value) {
 3   // ...
 4   var node =
 5   new SplayTree.Node(key, value);
 6   if (key > this.root .key) {
 7     node.left = this.root ;
 8     node.right = this.root_.right;
 9     this.root_.right = null;
10   } else {
11     node.right = this.root ;
12     node.left = this.root_.left;
13     this.root_.left = null;
14   }
15   this.root_ = node;
16 };
```

```
16 SplayTree.prototype.remove =
17 function(key) {
18   // ...
19   if (!this.root_.left) {
20     this.root_ =
21            this.root_.right;
22   } else {
23     // ...
24   }
25 };
```

Fig. 5. splay.js: The unordered addition of fields “left” and “right” in function `insert` will deoptimize the function `remove` at Line 16.

Case 1: splay.js The splay.js program implements the splay tree data structure, which is primarily designed for testing the performance of memory management. JSweeter finds an obscured performance issue caused by the underscored statements in function `insert` as shown in Figure 5. There is an instance of the typical *inconsistent field ordering* problem, where the fields “left” and “right” are added to the “SplayTree.Node” objects in different orders. As a consequence, when these “SplayTree.Node” objects are accessed, they would generate PICs and incur additional type checking overhead.

```
 1 h.prototype.SolveTOI =
 2 function(a) {
 3   // ...
 4   for ( ; ; ) {
 5     // ...
 6     if (b.m_flags & l.
           e_toiFlag)
 7       c = b.m_toi;
 8     else {
 9       // ...
10       b.m_toi = c;
11     }
12   }
13 };
```

Fig. 6. The simplified code for adding the field "m_toi" in box2d.js.

```
A.prototype.GetNext =
function () {
     return this.m_next
};
A.prototype.GetFixtureA =
function () {
     return this.m_fixtureA
};
A.prototype.GetFixtureB =
function () {
     return this.m_fixtureB
};
```

Fig. 7. Functions that are deoptimized by adding field "m_toi" in `h.SolveTOI`.

Even worse, these objects would deoptimize the `remove` function through the IC site at Line 16. And the consequent performance degradation incurred by using un-optimized version of `remove` would be prominent, because splay.js frequently inserts and removes nodes from the splay tree. Simply adding the fields "left" and "right" in the two conditional branches in the same order would fix this problem. A better solution is proactively adding both "left" and "right" in the constructor `SplayTree.Node`, which also avoids the problems caused by the `SplayTree.Node` objects in other places. We obtain 23% more scores from this simple fix.

Case 2: box2d.js The box2d.js program is a popular 2D physics engine. It has nearly 9500 lines of deminified code. Since **box2d.js** is compiled from Emscripten [2], a C++ to JavaScript compiler, it is full of simply-named variables such as "a", "Q", and *etc.*. Thus, finding performance issues manually for box2d.js is almost impossible even for an experienced programmer. With the help of JSweeter, we successfully fix three performance bugs.

Among the three bugs, one would incur deoptimizations for seven functions by adding a field "m_toi". This field addition operation is performed in function `h.SolveTOI`. We show a simplified version of `h.SolveTOI` in Figure 6, where we highlight the two access sites for field "m_toi": Line 7 is a read site and Line 10 is a write site. Line 10 changes the type of the objects referenced by "b", which deoptimize quite a few functions, such as those in Figure 7.

JSweeter outputs a **addFlds** hint to suggest adding the *m_toi* field in an early stage. In the bug report, JSweeter locates function `Oa` as the constructor of the objects pointed by *b* and the corresponding "Oa" object is in the function `z.Create`. However, our first attempt by directly adding the field *m_toi* followed by the creation of "Oa" object in `z.Create` does not eliminate the performance issue. A further investigation with the calling context information shows that the fields of object "Oa" fields are added in functions `A.Reset` and `A.b2Contact`. At this place, JSweeter cannot offer more help. Based on our human study of

[2] `https://github.com/kripken/emscripten`

```
1  GameBoyCore.prototype.initializeTiming = function () {
2    // ...
3    this.CPUCyclesTotal = (this.baseCPUCyclesPerIteration - this.
         CPUCyclesTotalRoundoff) | 0;
4  }
5  GameBoyCore.prototype.audioUnderrunAdjustment = function () {
6    // ...
7    this.CPUCyclesTotalCurrent += (underrunAmount >> 1)*this.machineOut;
8  }
9  GameBoyCore.prototype.iterationEndRoutine = function () {
10   // ...
11   this.CPUCyclesTotalCurrent += this.CPUCyclesTotalRoundoff;
12 }
13 GameBoyCore.prototype.recalculateIterationClockLimit = function () {
14   // ...
15   this.CPUCyclesTotal = this.CPUCyclesTotalBase + this.
         CPUCyclesTotalCurrent - endModulus;
16   this.CPUCyclesTotalCurrent = endModulus;
17 }
```

Fig. 8. All places that write to "CPUCyclesTotal" and "CPUCyclesTotalCurrent".

functions near to `A.Reset`, we realize `A.Update` is the best place to add the field *m_toi*. With this refactoring, all the seven deoptimizations are eliminated.

Case 3: gbemu.js The gbemu.js program is a GameBoy emulator. Unlike box2d.js, which allocates many empty objects and incrementally updates them, gbemu.js uses a big monolithic data structure named gameboy to store the virtual machine states. In this flat design, almost all the fields of gameboy are added by the constructor GameBoyCore and most of these fields are integers.

One representative issue is caused by the integer overflow of two fields: *CPUCyclesTotal* and *CPUCyclesTotalCurrent*. From their names, we guess these fields store the number of CPU cycles elapsed on the emulated CPU. There are only four places that write to *CPUCyclesTotal* and *CPUCyclesTotalCurrent* other than the constructor, summarized in Figure 8.

Taking *CPUCyclesTotal* as an example, its value can exceed 2^{30} at Line 15 of Figure 8, which is the upper bound for the small integer representation used by V8. The integer overflow triggers a representation change to use double value for *CPUCyclesTotal*. As a consequence, all fields in the object "gameboy" are lifted to double representations [5], and all operations related to these fields are impacted. As suggested by the factorOut hint, we use a separate object to place the *CPUCyclesTotal* and *CPUCyclesTotalCurrent* fields. In this way, all fields are not mutually impacted.

The second issue is that the field *mixerOutputCache* occasionally gets **NaN** via the computation as shown in Figure 9. Since JSweeter does not track the value flows, we cannot understand how *mixerOutputCache* becomes **NaN**. We simply add a **NaN** checking before assigning the computation result to *mixerOutputCache*.

JSweeter also outputs a **factorOut** suggestion for an anonymous closure assigned to array "LINECONTROL", which is responsible for screen rendering. In this case, large volume of closure instances are created and they deoptimize 163 times, where 90.4% of the deoptimizations are contributed by the field-access

```
GameBoyCore.prototype.mixerOutputLevelCache = function () {
  this.mixerOutputCache =
  ((((this.channel1currentSampleLeftTrimary +
  this.channel2currentSampleLeftTrimary +
  this.channel3currentSampleLeftSecondary +
  this.channel4currentSampleLeftSecondary) *
  this.VinLeftChannelMasterVolume) << 9) +
  ((this.channel1currentSampleRightTrimary +
  this.channel2currentSampleRightTrimary +
  this.channel3currentSampleRightSecondary +
  this.channel4currentSampleRightSecondary) *
  this.VinRightChannelMasterVolume));
}
```

Fig. 9. The unique place that writes to “mixerOutputCache”.

```
1 this.LINECONTROL[line] =
2 function (parentObj) {
3   if (parentObj.LCDTicks<80) {
4     // ...
5   }
6 }
```

Fig. 10. The IC site (highlighted area) that contributes most to the deoptimization of LINECONTROL.

```
function entry0(parentObj) {
  var ticks = parentObj.LCDTicks;
  processLT143(ticks, parentObj);
}
function
processLT143(ticks, parentObj) {
  if (ticks < 80) {
    // ...
  }
}
this.LINECONTROL[line] = entry0;
```

Fig. 11. Isolate the deoptimization site with other parts in LINECONTROL.

site at Line 3 of Figure 10. To factor out this problematic IC site, we take a two-step solution. We first define function `entry0` that only reads the field *LCDTicks* and keep other statements in function `processLT143`. Second, we add a tail call to `processLT143` in `entry0`, as shown in Figure 11. By this refactoring, we assign the unique instance of `entry0` to all the elements of array “LINECONTROL”, and this *frequent closures creation* problem is solved.

6 Related Work

JavaScript Performance Debugging. The most relevant work to us is Gong *et.al.*’s JITProf [9]. This work also performs a pattern matching based dynamic analysis to locate the code that causes JIT failures and results in performance degradation. However, JSweeter is more general and powerful than JITProf in four ways:

1. Our 6 bug patterns are not ad-hoc: Type mutation is their coherent reason to cause performance issues. This deep insight can guide programmers to find new bug patterns easily. Moreover, we also performed an empirical study and showed the pervasiveness of the proposed bug patterns. In contrast, JITProf only lists 7 bug patterns without explaining where these patterns come from.

2. Central to our algorithm is the type evolution graph (TEG), which is a uniform representation for different pattern matching algorithms. In contrast, JITProf designs individual pattern matching algorithm for each bug pattern, which precludes adding new patterns easily.

3. TEG aggregates the type information for sibling objects while JITProf traces the state for each individual object. TEG is superior for bug detection because, by contrasting the behavior of an object to its sibling objects, a deviated type evolution is more likely to be a real bug.

4. JSweeter is running offline and thus have more flexibility to run complicated pattern matching algorithms without incurring runtime overhead. For example, Algorithm 5 retrospects the historic type information to confirm the *partially initialized objects* bug. In contrast, JITProf works totally online and performs limited checks to decide a bug. Nevertheless, JITProf already incurs 18× runtime overhead even with events sampling.

Performance debugging on statically typed languages. Most of the works still rely on function execution time profiling data and statistical algorithms [20, 23, 18, 25, 3, 24, 8, 13]. However, as we argued, type mutations cannot be captured by time profiling results. The works Sherlog [26] and G2 [11] share similarity to ours. Sherlog infers a possible control flow from program start to the symptom site. The control flow information is useful for functional bugs, but it is unknown how performance bugs can benefit from it. Instead, JSweeter generates an object centric view of the type update process that only contains the operations pertaining to performance issues. The work G2 also models the log events as a graph and it backwardly and forwardly to search the root cause. Compared to G2, JSweeter goes further to generate refactoring suggestions by pattern matching the objects evolution history to our empirical observations.

Avoid type instability with type prediction. Instead of preventing type mutations, improving the type prediction successful rate can also speed up JavaScript execution. Hackett *et al.* are the first to design a type-inference algorithm that works for full JavaScript features [12], by performing type inference online with the help of type-feedback. In contrast, Kedlaya *et al.* use the type-inference to aid type-feedback to intelligently place type profiling hooks [17]. Santos *et al.* [21] developed a technique to generate a specialized version of the function for every combination of the parameter values for that function, which significantly enforces the power of constant propagation and other optimizations. All these works are orthogonal to ours, because our aim is involving programmers to address the performance bugs with complex logics.

7 Conclusion and Future Work

In this paper, we propose a dynamic analysis to detect, infer, and refactor six JavaScript performance issues incurred by type mutations. We first empirically study the performance bug patterns common in real world programs. Based on the study, we design a technique that analyzes the type evolution graph to infer the occurrence of the predefined code smells and synthesize refactoring suggestions. We implement a tool JSweeter and find nineteen performance bugs in Octane benchmark suite. These bugs can be effectively fixed by following JSweeter's refactoring suggestions and the benchmark scores for bug fixed programs can increase up to 23%.

References

1. https://asmjs.org
2. https://people.mozilla.org/~jorendorff/es6-draft.html
3. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: SOSP (2003)
4. Ahn, W., Choi, J., Shull, T., Garzarán, M.J., Torrellas, J.: Improving javascript performance by deconstructing the type system. PLDI (2014)
5. Bolz, C.F., Diekmann, L., Tratt, L.: Storage strategies for collections in dynamically typed languages. In: OOPSLA (2013)
6. Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., Tip, F.: Tool-supported refactoring for javascript. In: OOPSLA (2011)
7. Flückiger, O.: Compiled Compiler Templates for V8. Master's thesis (2014)
8. Fu, Q., Lou, J.G., Wang, Y., Li, J.: Execution anomaly detection in distributed systems through unstructured log analysis. In: ICDM (2009)
9. Gong, L., Pradel, M., Sen, K.: Jitprof: Pinpointing jit-unfriendly javascript code. In: Proc. ESEC/FSE. pp. 357–368. ACM (2015)
10. Gudeman, D.: Representing type information in dynamically typed languages (1993)
11. Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., Zhou, L.: G2: a graph processing system for diagnosing distributed systems. In: USENIXATC (2011)
12. Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for javascript. In: PLDI (2012)
13. Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T.: Performance debugging in the large via mining millions of stack traces. In: ICSE (2012)
14. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: ECOOP (1991)
15. Hölzle, U., Ungar, D.: Optimizing dynamically-dispatched calls with run-time type feedback. In: PLDI (1994)
16. Hölzle, U., Ungar, D.: A third-generation self implementation: Reconciling responsiveness with performance. In: OOPSLA (1994)
17. Kedlaya, M.N., Roesch, J., Robatmili, B., Reshadi, M., Hardekopf, B.: Improved type specialization for dynamic scripting languages. In: DLS (2013)
18. Liu, X., Mellor-Crummey, J.: Pinpointing data locality problems using data-centric analysis. In: CGO (2011)
19. McCutchan, J.: Accelerating oz with v8: Follow the yellow brick road to javascript performance. Google I/O Conference (2013)
20. Nistor, A., Song, L., Marinov, D., Lu, S.: Toddler: detecting performance problems via similar memory-access patterns. In: ICSE (2013)
21. Santos, H.N., Alves, P., Costa, I., Quintao Pereira, F.M.: Just-in-time value specialization. In: CGO (2013)
22. Van Cutsem, T., Miller, M.S.: Traits.js: Robust object composition and high-integrity objects for ecmascript 5. PLASTIC (2011)
23. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: profiling copies to find runtime bloat. In: PLDI (2009)
24. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: SOSP (2009)
25. Yan, D., Xu, G., Rountev, A.: Uncovering performance problems in java applications with reference propagation profiling. In: ICSE (2012)
26. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: Sherlog: error diagnosis by connecting clues from run-time logs. In: ASPLOS XV (2010)