Hidden Inheritance: An Inline Caching Design for TypeScript Performance

ZHEFENG WU, Alibaba Group, China
ZHE SUN, Alibaba Group, China
KAI GONG, Alibaba Group, China
LINGYUN CHEN, Alibaba Group, China
BIN LIAO, Alibaba Group, China
YIHUA JIN, Alibaba Group, China

TypeScript is a dynamically typed language widely used to develop large-scale applications nowadays. These applications are usually designed with complex class or interface hierarchies and have highly polymorphic behaviors. These object-oriented (OO) features will lead to inefficient inline caches (ICs) or trigger deoptimizations, which impact the performance of TypeScript applications.

To address this problem, we introduce an inline caching design called hidden inheritance (HI). The basic idea of HI is to cache the static information of class or interface hierarchies into hidden classes, which are leveraged to generate efficient inline caches for improving the performance of OO-style TypeScript programs. The HI design is implemented in a TypeScript engine STSC (Static TypeScript Compiler) including a static compiler and a runtime system. STSC statically generates hidden classes and enhanced inline caches, which are applied to generate specialized machine code via ahead-of-time compilation (AOTC) or just-in-time compilation (JITC). To evaluate the efficiency of this technique, we implement STSC on a state-of-the-art JavaScript virtual machine V8 and demonstrate its performance improvements on industrial benchmarks and applications.

CCS Concepts: • **Software and its engineering** → **Classes and objects**; *Dynamic compilers*; **Polymorphism**.

Additional Key Words and Phrases: TypeScript, AOTC, JITC, JavaScript, STSC, Hidden Classes, Inline Caches

ACM Reference Format:
Zhefeng Wu, Zhe Sun, Kai Gong, lingyun Chen, Bin Liao, and Yihua Jin. 2020. Hidden Inheritance: An Inline Caching Design for TypeScript Performance. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 174 (November 2020), 29 pages. https://doi.org/10.1145/3428242

1 INTRODUCTION

TypeScript [Microsoft 2014] is an open-source programming language. It is a syntactical superset of JavaScript and provides optional static types. As a syntactical superset of JavaScript, TypeScript preserves the semantics of JavaScript. It provides robust program abstractions, such as classes and interfaces, for developing large-scale JavaScript applications [Egret-3d 2017; IDE-VSCode 2017; NativeScript 2017; Superpowers 2018] and frameworks [Angular2 2017; Vue.js 2019]. Unfortunately, the type system of TypeScript is intentionally unsound for pragmatic reasons, and its static types

Authors' addresses: Zhefeng Wu, Alibaba Group, China, zhefeng.wu@alibaba-inc.com; Zhe Sun, Alibaba Group, China, zhe.sunz@alibaba-inc.com; Kai Gong, Alibaba Group, China, kai.gong@alibaba-inc.com; lingyun Chen, Alibaba Group, China, lingyun.cly@alibaba-inc.com; Bin Liao, Alibaba Group, China, bin.liao@alibaba-inc.com; Yihua Jin, Alibaba Group, China, yihua.jyh@alibaba-inc.com.

2475-1421/2020/11-ART174
https://doi.org/10.1145/3428242

are only used for static type checking instead of improving the execution performance. For a TypeScript application, its code is transpiled to plain JavaScript that then runs on a stock JavaScript virtual machine. Modern commercial JavaScript virtual machines (VMs) like V8 run the transpiled JavaScript code with its interpreter that collects types and profiling information. Then, their JIT compiler compiles hot functions to specialized machine code with the collected type information.

According to a series of investigations [Frederickson 2015; Stackoverflow 2017; Tiobe 2020], TypeScript has become one of the most popular languages by 2020. To take advantage of TypeScript, many existing JavaScript applications and frameworks are ported to TypeScript by adding static types. In addition to port JavaScript programs, some large-scale applications [IDE-VSCode 2017; Microsoft 2014; Superpowers 2018] and frameworks [Angular2 2017; NativeScript 2017; Vue.js 2019] are developed with TypeScript from scratch. They usually adopt the OO style programming pattern of TypeScript and heavily use interfaces and classes. Therefore, TypeScript introduces not only optional static types but also OO-style programming patterns to JavaScript world. As the popularity of TypeScript grows, its performance becomes a meaningful and significant topic.

```
1  interface I1 { x: number }
2  interface I2 extends I1 { y?: number }
3
4  function foo(i: I1) {
5    i.x++;
6  }
7
8  let a: I1 = {x: 1}
9  let b: I2 = {y: 2, x: 3}
10
11 foo(a);
12 foo(b);
```

(a)

```
1  class A {
2    x: number;
3    constructor(x: number) {
4      this.x = x;
5    }
6  }
7  class B extends A {
8    y: number;
9    constructor(x: number, y: number) {
10     super(x);
11     this.y = y;
12   }
13 }
14 let w = new A(1);
15 let h = new B(1, 2);
```

(b)

Fig. 1. Examples of typical OO-style TypeScript programs. (a) Polymorphic types of the parameter i in line 5. (b) Polymorphic types of the implicit parameter this in line 4.

Type polymorphism is prevalent in OO-style TypeScript programs. In the two examples of Figure 1, the operation of loading property x from parameter i (line 5 in Figure 1 (a)) needs to use type dispatching to match the type of i and then identify the location of x inside the object referenced by i. Similarly, the operation in line 4 of Figure 1(b) also needs to determine the type of *this* before adding a property x. Such type polymorphism will degrade the performance of inline caching, which is a key optimization adopted by modern JavaScript VMs (e.g., V8 [Google-V8 2019], ChakraCore [Microsoft 2018] and JSC [Apple 2018]) to accelerate execution. It is because the inline caching (IC) technique is based on the type stability that is violated in the TypeScript programs with complex interface and class hierarchies. Although the PIC (*Polymorphic Inline Cache*) technique proposed by Holzle et al [Hölzle et al. 1991] can improve the performance by introducing a dynamic search in the cache history, it is not as efficient as monomorphic inline caches. Besides, due to the limited capacity of the cache of PIC, cache overflows will lead to IC misses. Especially, in the contexts of JITC and AOTC, IC misses will trigger deoptimizations at runtime and impact the performance.

The goal of this paper is to reduce the IC misses caused by type polymorphism and improve the performance of OO-style TypeScript programs. We observe that the static types of TypeScript provide opportunities to solve the aforementioned IC miss issue. For instance, the inheritance information of the interfaces (I1 and I2) and classes (A and B) in Figure 1 imply a certain layout consistency: objects a and b possess a common property x, and objects w and h inherit a property

x from their common superclass. This layout consistency indicates a certain similarity between TypeScript and statically typed languages like C++ and Java.

Based on these observations, we propose an inline caching design called hidden inheritance (HI). The main idea of HI is to leverage the static information of class and interface hierarchies into statically generated hidden classes and generate enhanced inline caches. We implement the HI design in a TypeScript engine called STSC (Static TypeScript Compiler), which is built on TypeScript Compiler (TSC) [Microsoft 2014] and a commercial JavaScript VM V8 [Google-V8 2019]. STSC contains a compiler that statically compiles TypeScript programs into bytecode or executable machine code, and a runtime that loads the output. Further, to evaluate the efficiency of the HI technique, we run industrial benchmarks and real-world applications with STSC and V8. The evaluation shows that our HI design can significantly reduce the IC misses and improve the performance of OO-style TypeScript programs.

In summary, this paper makes the following contributions:

(1) It proposes the idea of hidden inheritance, which extends conventional hidden classes and inline caching with OO hierarchy technique.
(2) It presents a TypeScript engine design (STSC), which exploits the static types of TypeScript with HI technique.
(3) It implements STSC on top of TSC, the TypeScript Compiler, and V8, a commercial JavaScript VM.
(4) It provides a detailed evaluation of STSC's performance on industrial benchmarks and applications.

This paper is organized as follows; Section 2 gives background; Section 3 introduces the overview; Section 4 defines hidden inheritance relationship; Section 5 presents the implementation details of the STSC engine; Section 6 discusses some limitations of STSC; Section 7 evaluates its performance; Section 8 covers related work; Section 9 summarizes this paper and proposes future work.

2 BACKGROUND

Hidden classes (HC) and inline caching (IC) [Chambers et al. 1989; Deutsch and Schiffman 1984; Hölzle et al. 1991] are two fundamental techniques used to improve the performance of dynamically typed languages. They are key optimizations for modern commercial JavaScript VMs, which usually generate hidden classes and inline caches during the initial warmup phase and then feed them to JITC compilers to emit efficient code.

2.1 Hidden Classes

The basic idea of hidden classes is similar to the map in Self [Chambers et al. 1989] and more introductions can be found in literature [Artoul 2015; Bevenius 2018; Chambers et al. 1989; Choi et al. 2019; Serrano and Feeley 2019]. Here we briefly describe the idea of hidden classes. To represent the layouts of objects at runtime, each object is assigned to a data structure called hidden class. Objects with the same layout can share a common hidden class, which helps to enable the optimizations of JavaScript compilers. For instance, compilers can produce efficient code for accessing object properties since the layout information of the objects can be obtained through their hidden classes. Throughout this paper, we only focus on the hidden class design of V8, since the designs in other popular JavaScript VMs (e.g., JSC [Apple 2018] and Chakra [Microsoft 2018]) are similar.

Figure 1 shows two typical OO-style patterns of TypeScript applications. First, the code in Figure 1 (a) defines a function *foo*, which increases the value of the property x of parameter i. Two objects *a* and *b* are created and passed to the function *foo*. Figure 2(a) illustrates the hidden class mechanism of V8. When an object *a* is created (in line 8), its hidden class field points to an initial hidden class

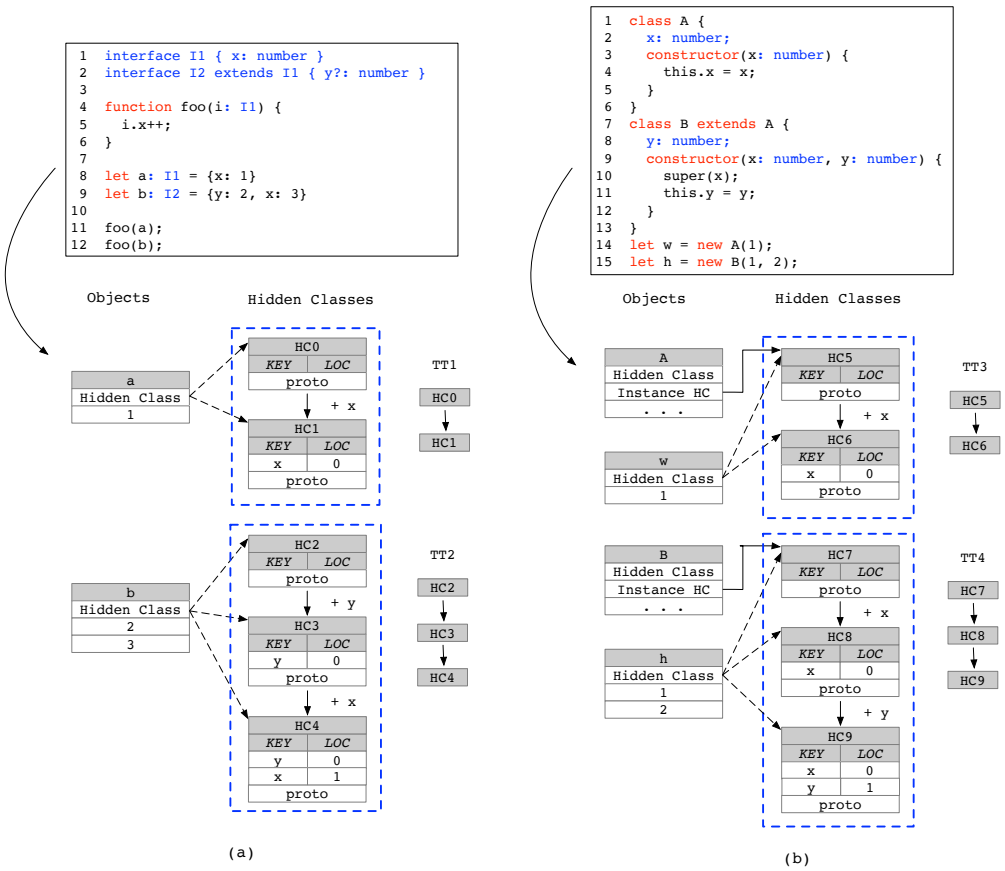


Fig. 2. The hidden class design of V8. (a) The generated hidden classes and transition trees when allocating objects, *a* and *b*. (b) The generated hidden classes and transition trees when allocating objects, *w* and *h*.

HC0. The HC0 contains no property layout descriptions since no property is added. Next, when a property *x* is added to the object *a* inside the function *foo*, a new hidden class HC1 is created by V8 runtime to record the new property's layout information in the form of a key-location pair. After HC1 is created, the hidden class field of the object a is reset to HC1 from HC0, and the value of the property *x* is stored to the location 0 inside *a*. To reuse hidden classes and reduce IC misses, V8 links these hidden classes HC0 and HC1 to form a transition tree denoted as TT1. With transition trees, the overhead of creating hidden classes can be avoided when the same properties are added next time. Another object *b* is created and initialized similarly. It yields another transition tree TT2, which contains three hidden classes: HC2, HC3, and HC4.

Instance Hidden Classes. Consider the code in Figure 1 (b). In TypeScript, the class syntax is syntactic sugar, and the two classes A and B are represented with their constructor functions A and B, respectively. Therefore, when we talk about a class in this paper, we mean its constructor function. Each function in V8 has two hidden classes, one for itself and the other for the objects created by the function. We use the term *instance hidden class* (IHC) to represent the latter. When the runtime executes the code of class declarations, the corresponding constructor functions and associated IHCs are allocated. For instance, the IHCs of classes A and B are hidden classes HC5 and

HC7, respectively. When the execution reaches line 14, object *w* is created with its hidden class field pointing to the IHC of class A, denoted as HC5. Next, when the code (in line 4) is executed, a property *x* is added to the object *w*. Similar to the previous discussion, the addition operation will cause the runtime to create a new hidden class HC6 and finally form a transition tree TT3. The initialization process (line 4) of object *h* is similar. Therefore, after executing the code (in line 15) of allocating and initializing the object *h*, another transition tree TT4 is created. At last, the hidden class of the object *h* becomes hidden class HC9.

Prototype Chains. Besides property layout information, V8 records prototype chain information in hidden classes and treats prototype chains as a part of hidden classes. The *proto* field of each hidden class points to the prototype object of the hidden class's owner objects. In this way, the objects with a common hidden class can share the properties defined on the common prototype objects. Besides, prototype chains are applied to implement the semantics of class inheritance for JavaScript. Figure 3 illustrates the structures of the constructor functions A and B in Figure 2 (b). By linking the two prototype objects *A.prototype* and *B.prototype* to form a prototype chain, the instance *h* of class B can access the functions of class A via traversing the prototype chain inside its hidden class.

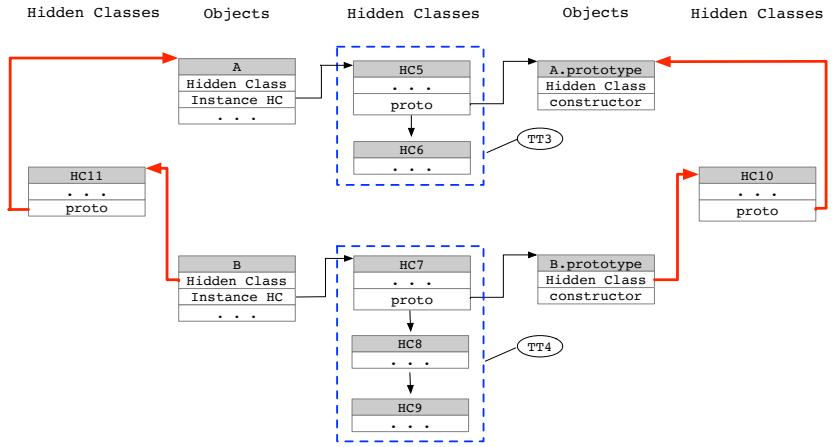


Fig. 3. Illustrating how V8 leverages prototype chain technique to implement class inheritance. The red arrows denote the two prototype chains starting from function B and its prototype.

2.2 Inline Caching

Inline caching is a fundamental optimization technique enabled by hidden classes. Modern JavaScript VMs heavily rely on IC optimization to improve execution performance. For an object access site, where an object property is loaded or stored, the basic idea of IC is to cache the hidden classes of incoming objects and the location information of target properties to accelerate the access. We use the term *incoming object* to denote the encountering object at the access site where its property is loaded or stored. Without IC optimization, the runtime system would be called at every object access site and perform the access operation at the cost of a series of time-consuming actions (e.g., looking up the target property on the hidden class or the prototype chain of the incoming object). To understand IC design, we consider two scenarios of object properties.

Local Properties. When the runtime encounters a new hidden class and identifies the location of the target property inside the object, it caches a pair <hidden class, location> to accelerate the next

loading operations. For instance, as shown in Figure 2 (a), the runtime would cache a pair <HC1, 0> (the hidden classes HC1 of object *a* and the location of property *x*) for the access site (in line 5) after calling the function *foo* (in line 11). If the code in line 11 is executed again, the runtime can directly load the value of property *x* at location 0 since the hidden class of the incoming object matches the cached hidden class HC1. Next, when executing the code (line 12), since the hidden class HC4 of object *b* is not identical to HC1, an IC miss occurs. In the IC miss routine, the runtime loads the value of *x* after identifying its location, and cache pairs <HC1, 0> and <HC4, 1> to accelerate the subsequent operations. In the IC design of V8, an object access site that only caches a single hidden class is called *monomorphic*; if it caches multiple hidden classes, it is called *polymorphic*. When the number of cached hidden classes for an object access site exceeds four, it is called *megamorphic*.

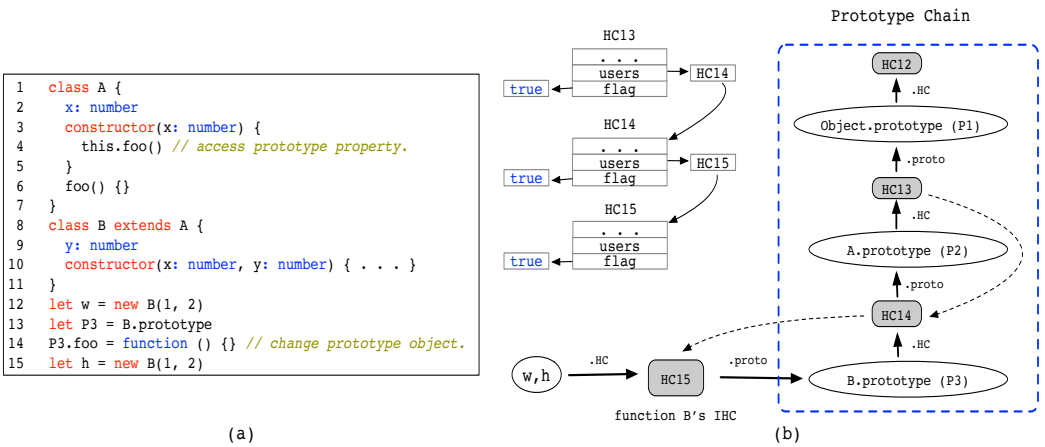


Fig. 4. An example of accessing prototype properties. (a) An example of changing prototype chains. (b) The double linked design of prototype chains in V8.

Prototype Properties. When accessing the properties on prototype chains, the ICs need to cache more information. Since the prototype chain between an incoming object and the owner object of the target property may be changed (e.g., adding new properties to the prototype objects on the prototype chain or breaking the prototype chain via corresponding built-in APIs), the generated ICs need to check not only the hidden class of the incoming object but also the stability of the original prototype chain. To efficiently detect the changes on prototype chains, V8 adds two data structures to hidden classes: *users* and *flag*. The *users* of a hidden class of a prototype object records the 'users' hidden classes that have their *proto* fields pointing to the prototype object; The *flag* object of a hidden class is used to indicate if the prototype chain has been changed. It wraps a boolean value to indicate the stability of the prototype chain.

For the program in Figure 4(a), the corresponding prototype chain relationship for classes A and B is shown in Figure 4(b). When the runtime executes the code in line 12, it caches a 4-tuple <HC15, *flag* of HC15, P2, 0> data and records hidden classes HC15 and HC14 to the *users* arrays of their prototype objects' hidden class HC14 and HC13, respectively. In the 4-tuple data structure, HC15 is the hidden class of the incoming object *w*; the *flag* object of HC15; the owner object P2 of the target property *foo*, and the location of *foo* inside P2. When the code in line 4 is executed again, the runtime first checks the hidden class of the incoming object and the boolean value of the *flag* object. Then, if both checks succeed, the runtime directly loads the value from the cached owner object P2 with location 0.

Next, when executing the code in lines 13-14, the runtime will change the layout of prototype object P3 by adding a new property *foo*, which will lead to a hidden class transition. In this scenario, the runtime reversely traverses the prototype chain (via the *users* array stored in hidden class HC14) to reset the boolean values of the *flag* objects of HC15 and HC14 to *false*. In this way, when the execution reaches line 15, an IC miss will be triggered in line 4, and the expected property *foo* on P3 is loaded by calling a runtime routine. Note that more scenarios can trigger the invalidation mechanism, such as deleting prototype properties or breaking prototype chains through a built-in API *setPrototypeOf()*.

3 OVERVIEW

The goal of this paper is to generate efficient inline caches and reduce IC misses by exploiting the static type information of OO-style TypeScript programs. To achieve this goal, we implement an inline caching design called hidden inheritance (HI). By leveraging the interfaces and classes inheritance information in static types, HI can produce efficient inline caches for the OO-style polymorphic operations. In this section, we present the main ideas of HI as listed in Table 1.

Table 1. Main ideas in Hidden Inheritance (HI).

Main Idea	Description
Mapping static types to hidden classes	Interface and class types are statically translated into hidden classes.
Representing inheritance relationships	Inheritance information is represented as the HI relationship between hidden classes.
HI-based inline caches	Leveraging HI relationship to emit efficient inline caches.
Maintaining and reconstructing HI relationships	Maintaining and reconstructing HI relationships to reduce IC misses at runtime.

Mapping Static Types to Hidden Classes. The first idea to translate static types (e.g., interfaces and classes) to hidden classes. It is motived by our observation that if a TypeScript program is passed static type checking, its static types can be trusted. Although the type system and type checking of TypeScript is unsound, the static types are proper candidates for producing hidden classes to specify object layouts. The type violations at runtime can be detected by inserting runtime checks. By statically generating hidden classes from static types, the overheads of collecting types (e.g., hidden class generations and transitions) can be reduced. Besides, it also provides the essential type information for AOTC and JITC to produce optimized machine code.

Representing Inheritance Relationships. The second idea is that the inheritance design in conventional statically typed languages (e.g., C++ and Java) can be leveraged to extend the conventional hidden class design. We define a relationship between the statically generated hidden classes called *hidden inheritance relationship* (HI relationship) based on the lexical inheritance information. The HI relationship specifies structural compatibility between hidden classes. It specifies the object layout of a superclass is a prefix of its derived classes. Besides, it also eliminates the name shadowing between the local properties and prototype properties to facilitate emitting efficient inline caches.

HI-based Inline Caches. The third idea is to emit HI-based inline caches for accessing the instances of each class/interface and its derived classes/interfaces. Based on the features of the HI relationship (e.g., the structural compatibility of object layouts and no name shadowing), monomorphic inline caches can be statically emitted to handle the polymorphic operations in OO-style programs. The key difference between HI-based inline caches and conventional inline caches is that the former not only matches the cached hidden classes and the incoming hidden classes but also detects the HI relationship between them.

Maintaining and Reconstructing HI Relationships. The last idea is to maintain and reconstruct HI relationships at runtime. We notice that dynamic operations (e.g., property additions and deletions)

might violate the original HI relationships. For instance, the operation of adding a new property would trigger hidden class transitions. However, a new HI relationship can be built between the original hidden classes and the newly created hidden classes if they satisfy the conditions of the HI relationship. In this way, the newly created hidden classes can still pass the type checks of the HI-based inline caches. Therefore, maintaining and reconstructing HI relationships can reduce IC misses when dynamic operations take place at runtime.

4 HIDDEN INHERITANCE RELATIONSHIP

The key components of the HI technique are the concepts of *hidden inheritance relationship* (HI relationship) and *hidden inheritance tree* (HI tree). The HI relationship defines an enhanced structural subtyping relationship between hidden classes. After defining the HI relationship, we define the HI tree which consists of the hidden classes with HI relationships. In this section, we first classify properties for hidden classes. Then, we present the definitions of the HI relationship and HI tree.

4.1 Defining Properties for Hidden Classes

To define the HI relationship, we extend the classification of the *local property* and *prototype property* in Section 2.2 to hidden classes. Recall the description of hidden classes in Section 2.1. An object's layout information and the prototype chain pointed by the field *proto* in its hidden class are stored in the hidden class of the object, so we can define the same concepts of *local property* and *prototype property* for hidden classes. For example, HC9 in Figure 2 (b) contains two local properties, x and y, and two prototype properties, the two *constructor* functions, while HC7 contains no local properties and has the same prototype properties as HC9. After defining the two concepts, we introduce *lookup table (vtable)*, which is a table that is used to cache prototype properties information. Each vtable consists of 4-tuple <name, type, owner, loc> entries. For each prototype property, its vtable entry records its property name, its type (e.g., function, getter or setter), its owner object, and its location inside the owner object. We call the properties cached in vtables as *vtable properties* for the rest of this paper.

Table 2. Three sets are defined for hidden class H.

Set	Definition
local(H)	The set of the local properties (in the form of 3-tuple <name, loc, type>) of H.
on_proto(H)	The set of the prototype properties (in the form of 4-tuple <name, type, owner, loc>) of H.
vtable(H)	The lookup table (in the form of 4-tuple <name, type, owner, loc>) for on_proto(H) considering name shadowing.

To simplify our discussion, we define three sets for hidden classes (in Table 2), where the symbol H represents an arbitrary hidden class. If we compute these sets for the hidden classes of the transition trees in Figure 2 (b), we can obtain the following results in Table 3. In Table 3, we can see that the hidden classes in the same transition tree contain different local properties but share the same prototype properties. Note that, in this section, we ignore the properties on the default prototype object *Object.prototype* to simplify our description and computation. Especially, note that as a kind of lookup table, a hidden class's vtable may not cache all prototype properties. For instance, in Table 3, the *constructor* of class A is not cached in the vtables of HC7, HC8, and HC9 after considering name shadowing.

Table 3. The results of computing the three sets for the hidden classes in Figure 2 (b).

HCs / Sets	local	on_proto	vtable
HC5	∅	{ <constructor, function, A.prototype, 0>}	{ <constructor, function, A.prototype, 0>}
HC6	{ <x, 0, number>}	{ <constructor, function, A.prototype, 0>}	{ <constructor, function, A.prototype, 0>}
HC7	∅	{ <constructor, function, A.prototype, 0>, <constructor, function, B.prototype, 0>}	{ <constructor, function, B.prototype, 0>}
HC8	{ <x, 0, number>}	{ <constructor, function, A.prototype, 0>, <constructor, function, B.prototype, 0>}	{ <constructor, function, B.prototype, 0>}
HC9	{ <x, 0, number>, <y, 1, number>}	{ <constructor, function, A.prototype, 0>, <constructor, function, B.prototype, 0>}	{ <constructor, function, B.prototype, 0>}

4.2 Hidden Inheritance Relationship

After classifying properties and defining the property sets for hidden classes, we can define the inheritance relationship between hidden classes. The hidden inheritance relationship is a relationship between two hidden classes A and B denoted as $A \lhd B$, if and only if they satisfy the following four conditions:

$$A \lhd B \Leftrightarrow \begin{cases} 1\text{: local(A)} \subseteq \text{local(B)} \\ 2\text{: vtable}_{NT}\text{(A)} \subseteq \text{vtable}_{NT}\text{(B)} \\ 3\text{: local}_{N}\text{(A)} \cap \text{vtable}_{N}\text{(B)} = \emptyset \\ 4\text{: vtable}_{N}\text{(A)} \cap \text{local}_{N}\text{(B)} = \emptyset \end{cases}$$

Fig. 5. The definition of hidden inheritance relationship for hidden classes A and B.

The HI relationship can be regarded as an enhanced structural subtyping relationship. The first condition enforces the layout compatibility between A and B, which means local(A) is a subset of local(B). The second condition requires the subset relationship between the vtables in terms of their property names (denoted by subscript N) and property types (denoted by subscript T). The last two conditions eliminate the name conflicts between local properties and vtable properties. The subscript N denotes the sets of names.

The first two conditions are designed for specifying the consistency of local properties and vtable properties between hidden classes, which is essential for generating monomorphic inline caches. The last two conditions are designed to statically specify the locations of properties and remove the overhead of traversing prototype chains in inline caches. From the definition of HI relationship, we can prove the following two corollaries.

(1) If $A \lhd B$, then $A \lhd A$. Since the first two conditions are obvious, we only need to prove the last two conditions, which are the same condition: $\text{local}_N\text{(A)} \cap \text{vtable}_N\text{(A)} = \emptyset$. Since $A \lhd B$, then by definition we have $\text{local}_N\text{(A)} \cap \text{vtable}_N\text{(B)} = \emptyset$ and $\text{vtable}_{NT}\text{(A)} \subseteq \text{vtable}_{NT}\text{(B)}$, so we can deduce $\text{local}_N\text{(A)} \cap \text{vtable}_N\text{(A)} = \emptyset$.

(2) If $A \lhd B$ and $B \lhd C$, then $A \lhd C$. Because the first two conditions obviously satisfy transitivity, we only need to prove transitivity for the last two conditions. Note that we have $\text{local(A)} \subseteq \text{local(B)}$ and $\text{local}_N\text{(B)} \cap \text{vtable}_N\text{(C)} = \emptyset$, then $\text{local}_N\text{(A)} \cap \text{vtable}_N\text{(C)} = \emptyset$ holds. Similarly, note that we have $\text{vtable}_{NT}\text{(A)} \subseteq \text{vtable}_{NT}\text{(B)}$ and $\text{vtable}_N\text{(B)} \cap \text{local}_N\text{(C)} = \emptyset$, then $\text{vtable}_N\text{(A)} \cap \text{local}_N\text{(C)} = \emptyset$ holds. Therefore, the HI relationship satisfies transitivity.

4.3 Hidden Inheritance Tree

After defining the HI relationship, we define *hidden inheritance tree*, which consists of hidden classes as the tree nodes. A HI tree not only requires the HI relationships between each parent node and its child nodes, but also each leaf node and itself. For instance, a HI tree A1 ◁ (A2, A3) requires four HI relationships: A1 ◁ A2, A1◁ A3, A2 ◁ A2, and A3 ◁ A3. Note that we also have A1 ◁ A1 according to the first corollary. With the transitivity of the HI relationship, we can easily prove that each node in a HI-tree holds HI relationship with itself and all its descendant nodes in the same HI tree.

5 IMPLEMENTATION

In this section, we describe how to implement the HI design. We build a TypeScript engine called STSC on top of TSC and Google V8. STSC implements the main ideas of HI design in two phases, *compilation phase* and *runtime phase*. We first describe how to represent the HI relationship by extending the data structure of hidden classes. Next, we describe the detailed designs of the two phases with examples. Figure 6 shows the pipeline of STSC.

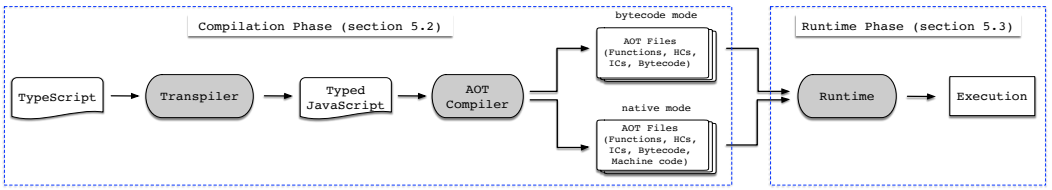


Fig. 6. The pipeline of STSC.

5.1 Representing HI Relationship

To represent a HI tree and accelerate the detecting of HI relationships, similar to the approach of [Cliff Click 2002], STSC adds two fields *level* and *supers* to hidden classes. The first field *level* represents the height of a node in its owner HI tree. The second field *supers* points an array that records the node itself and all the node's ancestors in its owner HI tree. Therefore, the HI relationship between two arbitrary hidden classes *A* and *B* can be efficiently detected by checking if *B.level* is no less than *A.level* and *A* is identical to *B.supers[A.level]*. In addition to these two fields, another field *vtable* is added to hidden classes to record the lookup tables for prototype properties.

5.2 Compilation Phase

STSC implements the first three ideas of HI (described in Section 3) in the compilation phase, including encoding static types, constructing HI trees from static types, computing inline caches, and generating code.

5.2.1 Transpiling TypeScript to Typed JavaScript. Since standard JavaScript VMs like V8 can not directly compile and execute TypeScript programs, STSC implements a transpiler (an extended version of TSC), which not only reuses the original passes of TSC to parse TypeScript code, infer types, and report type errors but also extracts static types and encodes these types into some JSON strings. We call these JSON strings as TAs (Type Annotations), which are embedded into the emitted JavaScript code in the form of comments. The emitted JavaScript code is called *typed JavaScript*. In typed JavaScript programs, each variable is decorated with a comment that encodes a type *id* to indicate its corresponding TA. In this way, the static type information of original TypeScript can be

preserved in the emitted typed JavaScript, which would be compiled by STSC in the next static compilation phase. Since the design of TA is not the main contribution of this paper, we present a more detailed description of TA in appendix A.

5.2.2 Generating HI Trees, Function Objects and Prototype Objects. After producing typed JavaScript code, STSC first decodes the embedded type information. Then, STSC compiles the typed JavaScript programs to ASTs (Abstract Syntax Trees) and attaches the decoded static types to the ASTs. Next, STSC visits the typed AST to generate HI trees, function objects, and prototype objects. To construct HI trees, STSC translates interfaces/classes to hidden classes and builds HI relationships between them. As for the four conditions of HI relationship, the first two conditions can be satisfied by stacking the local properties and vtable properties of a 'parent' hidden class to its 'child' hidden classes. This is similar to the object models in statically typed languages (e.g., C++ and Java). However, the last two conditions may be violated if we simply map all the properties in interfaces/classes to the local properties and the vtable properties of the corresponding hidden classes. Fortunately, we notice that this problem can be addressed by specially selecting the local properties and vtable properties when generating hidden classes. Therefore, the selection of properties is a key design of constructing HI trees. Besides, our another observation is that the construction process can be significantly simplified by pruning multi-inheritance to single inheritance. Based on these observations, we propose a construction algorithm, which consists of the following four steps.

(1) Select HI-candidates from class/interface types by the simplification strategy illustrated in Table 4, and linking the HI-candidates to form HI-candidate trees. Note that each node in the HI-candidate trees is a data structure corresponding to a class type or an interface type.

Table 4. Selecting HI candidates from static types using a pruning strategy.

Classes / Interfaces	HI Candidates
class A extends B {}	B < A
class A extends B implements I1, I2 {}	B < A
class A implements I1, I2 {}	I1 < A
interface I extends I1, A {}	I1 < I
class A {}	A
interface I {}	I

(2) Perform two top-down visits on the HI-candidate trees to select local properties and vtable properties for satisfying the last two conditions of the HI relationship. At the first pass of visiting, for each node A, perform the following two computations:
   (a) Compute A's local-set and vtable-set, which are initialized to the empty set at the beginning. If A is a class type, all its non-static properties and non-static member functions are added to its local-set and vtable-set respectively. If A is an interface type, all its properties are added to its local-set.
   (b) Compute and remove the name conflicting properties from local-sets and vtable-sets. STSC iterates over all the ancestors of A. For each ancestor B of A, STSC marks the properties Pi in A's local-set and the properties Fi in the vtable-sets of A or B, if the Pi and Fi have the same name. Similarly, STSC marks the properties Pi in B's local-set and the property Fi in A's vtable-set, if the Pi and Fi have the same name.

   After the first visiting has completed, perform another visit on the HI-candidate trees to remove the marked properties from the local-sets and vtable-sets.
(3) Perform a top-down breath-first visit on HI-candidate trees. For each node A, STSC merges A's local-set to A's immediate child nodes' local-sets and recomputes the locations of local

properties to satisfy the first condition of the HI relationship. Next, merge the A's vtable-set to its immediate child nodes' vtable-sets to compute the new vtable-sets and meet the second condition of HI relationship.

(4) Visit the HI-candidate trees and translate local-sets and vtable-sets into hidden classes and vtables. The associated member functions (e.g., constructor) and prototype objects are created as well. Next, STSC links the prototype objects of classes to form prototype chains and creates the associated backward links by filling the *users* array of the hidden classes of the prototype objects. At last, these generated prototype objects are set to corresponding hidden classes, and the hidden classes are linked to form HI trees.

The Correctness of HI Tree Construction. Here we discuss the correctness of the construction algorithm. Consider two hidden classes B and A in a HI tree constructed by the algorithm and assume that B is an immediate ancestor of A. Let us prove the four conditions of HI relationship for B and A. First, from the description of step 3, the relationship between B and A obviously satisfies the first two conditions of HI. Since the last two conditions are similar, we first prove the fourth condition. To prove $vtable_N(B) \cap local_N(A) = \emptyset$, we employ the proof by contradiction. Without loss of generality, we assume that a property $p \in local_N(A)$ and a property $f \in vtable_N(B)$ have the same name. According to step 3, the owner node of p (denoted as $owner_p$) and the owner node of f (denoted as $owner_f$) in the same HI-candidate tree should be one of the following three cases: (1) $owner_p$ is an ancestor of $owner_f$; (2) $owner_f$ is an ancestor of $owner_p$;(3) $owner_p$ is identical to $owner_f$. For anyone of these three cases, since the property p and property f have the same name, step 2(b) would mark and remove them from their owners' local-sets and vtable-sets. Therefore, such properties do not exist, and the fourth condition of HI relationship are satisfied for B and A. Similarly, the third condition of HI can be proved in the same way. Finally, the HI relationship holds between B and A. In addition, if B is identical to A, the HI relationship between B and A can be proved in a similar way. Therefore, each leaf node of the HI tree holds a HI relationship with itself. In conclusion, the trees constructed by the algorithm satisfy the definition of HI tree.

5.2.3 An Example of Constructing HI Trees. To understand the construction algorithm, we consider the program in Figure 7. The program declares two interfaces I1 and I2, and two classes A and B. STSC firstly selects three HI-candidates: I2, I1 < A, and A < B. Then, these HI-candidates are linked into two HI-candidate trees: I2 and I1 < A < B. Next, STSC computes the local-sets and vtable-sets for these four nodes. The process of computing of the local-sets and vtable-sets is illustrated in the following Table 5. Each row displays the contents of the local-sets and vtable-sets after each step. We can observe that the two entries, <y, 1, number> in the local-set of A and the <y, getter, B.prototype, 1> in the vtable-set of B, are removed in the step-2(b) since they violate the third condition of HI relationship. After processing name conflicts, in step-3, STSC visits the HI-candidate trees, merges the contents of local-sets and vtable-sets, and recomputes the locations of the local properties. At last, STSC creates four hidden classes, HC16, HC17, HC18, and HC19, from the final local-sets and vtable-sets, and link them to form two HI trees HI-tree1 and HI-tree2. Meanwhile, two functions A and B and their associated prototype objects are created. The two hidden classes HC17 and HC18 are installed into the functions A and B as their IHCs, and their *proto* fields are set to the corresponding prototype objects. Figure 7 shows the final HI trees and prototype objects.

HI Trees and Transition Trees. Comparing the HI tree in Figure 7 and the transition trees of V8 in Figure 2(d), we can observe several differences between them. Firstly, HI trees are statically defined by the static class/interface declarations, while transition trees are defined by actual execution paths at runtime. Secondly, the nodes of HI trees may come from different classes and each parent node maintains a HI relationship with itself and its descendant nodes. In comparison, the nodes of each transition tree transit from the IHC of a single class and share the same prototype chain.

Table 5. The contents of local-sets and vtable-sets after each step for the interfaces (I1 and I2) and classes (A and B) in the program of Figure 7.

	Sets	interface I2	interface I1	class A	class B
Step-2(a)	L	{<y, 0, number>}	{<x, 0, number>}	{<x, 0, number>, <y, 1, number>}	{<z, 0, number>}
	V	∅	∅	{<constructor, fun, A.prototype, 0>, <bar, fun, A.prototype, 0>}	{<constructor, fun, B.prototype, 0>, <y, getter, B.prototype, 1>}
Step-2(b)	L	{<y, 0, number>}	{<x, 0, number>}	{<x, 0, number>}	{<z, 0, number>}
	V	∅	∅	{<constructor, fun, A.prototype, 0>, <bar, fun, A.prototype, 0>}	{<constructor, fun, B.prototype, 0>}
Step-3	L	{<y, 0, number>}	{<x, 0, number>}	{<x, 0, number>}	{<x, 0, number>, <z, 1, number>}
	V	∅	∅	{<constructor, fun, A.prototype, 0>, <bar, fun, A.prototype, 1>}	{<constructor, fun, B.prototype, 0>, <bar, fun, A.prototype, 1>}

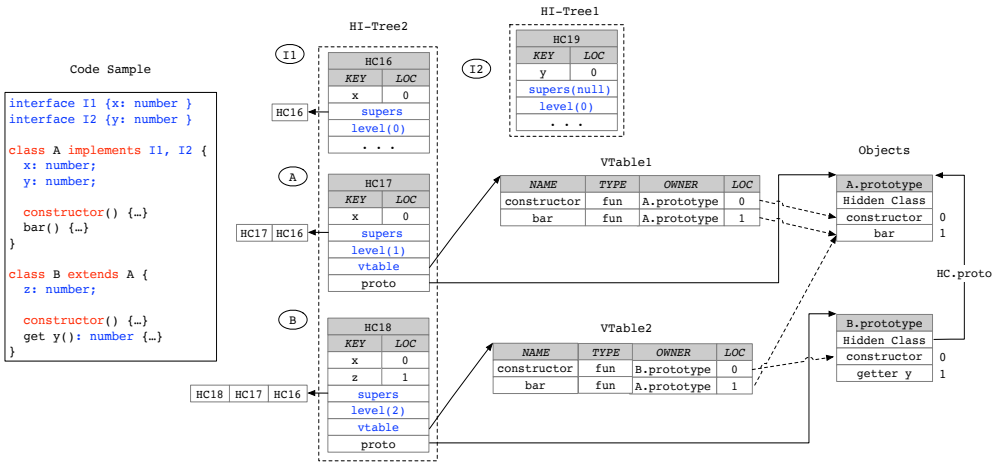


Fig. 7. An example of constructing a hidden inheritance tree. We create four hidden classes (HC16, HC17, HC18, and HC19) for the interfaces I1 and I2 and classes A and B. The HC17 and HC18 are used as the IHCs of the classes A and B, respectively.

5.2.4 Translating Object Allocations. After HI trees and function objects have been created, they are applied to optimize object allocations. As explained in Section 2.1, V8 assigns each newly created object with an empty IHC, and the object's properties are added until runtime execution. Unlike current V8 design, STSC statically generates non-empty hidden classes based on static types and uses them as the IHCs of corresponding constructor functions. STSC optimizes object allocations with the statically produced IHCs. By pre-adding properties in the IHCs, STSC can avoid the hidden class transitions due to property additions. In addition, to translate object allocation operations and maintain the semantics compatibility for JavaScript, *uninitialized properties* are introduced by STSC to implement the semantics of properties' existence. An uninitialized property is implemented by initializing the default value of the property to a unique STSC-internal value, *uninitialized marker (uninit-marker)*. This special value is used to indicate the absence of object properties at runtime. STSC combines this design of uninitialized properties with HI relationship to produce HI-based inline caches.

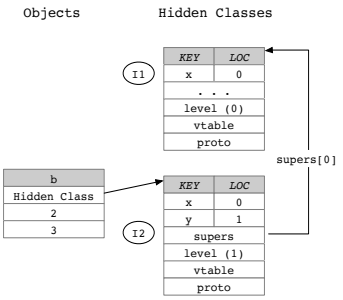
5.2.5 Generating HI-based Inline Caches. After creating hidden classes and HI trees, STSC computes HI-based inline caches for property access operations. Since the variables in TypeScript are annotated with types, STSC resolves the corresponding hidden classes of the types and computes HI-based inline caches based on the associated hidden classes. The key difference between HI-based inline caches and conventional inline caches is that the former not only matches the cached hidden classes and the incoming hidden classes like the latter but also detects the HI relationship between them. Based on the four conditions of HI relationship, HI-based inline caches can efficiently access the properties of the objects when their hidden classes satisfy a HI relationship, which is the key design of STSC to avoid IC misses. In this section, we elaborate the HI-based IC design for load and store operations.

```
// 'obj' is the incoming object at runtime.
// 'val' is the value of the property at runtime.
// 'key' is the property name.
// 'aot_hcs' is the set of statically generated hidden classes.

IRGen_POLY_IC_LoadProperty(aot_hcs, key) {
  foreach cached_hc of aot_hcs {
    if cached_hc.find(key) { // Check the existence.
      offset = cached_hc.getOffset(key)
      level = cached_hc.level
      // Check HI relationship.
      emit " supers = obj.HC.supers                                    "
      emit " if level < supers.length and cached_hc == supers[level] "
      if cached_hc.is_local(key) {
        // case 1: a local property.
        emit " val = obj[offset]              "
        emit " if val == uninit_marker        "
        emit "   val = undefined              "
        emit " goto __done                    "
      } else if cached_hc.is_getter_in_vtable(key) {
        // case 2: a getter on the prototype chain.
        emit " val = obj.HC.vtable[offset]    "
        emit " if val != undefined            "
        emit "   getter = val.owner[val.loc]  "
        emit "   val = call getter(obj)       "
        emit " goto __done                    "
      } else if cached_hc.is_in_vtable(key) {
        // case 3: a property on the prototype chain.
        emit " val = obj.HC.vtable[offset]    "
        emit " if val != undefined            "
        emit "   val = val.owner[val.loc]     "
        emit " goto __done                    "
      }
    }
  }
  emit " call Slow_Load "
  emit " __done:        "
}
```

(b)

```
interface I1 { x: number }
interface I2 extends I1 { y?: number }

function foo(i: I1) {
  i.x++;
}
. . .
let b: I2 = {y: 2, x: 3}
. . .
foo(b);
```

(a)



(c)

Fig. 8. (a) An example of loading properties in the program of Figure 1 (a), where the cached hidden class is created from interface I1. (b) STSC's code generation algorithm for loading properties. (c) STSC creates two hidden classes from two interfaces (I1 and I2) and builds a HI-tree.

Load Operations. Figure 8 (b) displays the code generation algorithm of STSC for loading properties. Assuming the hidden classes of the target type are recorded in a set *aot_hcs* and the key is the name of the target property. The algorithm iterates each hidden class and identifies the kinds and location of the target property and computes inline caches. When visiting a hidden class *cached_hc*, if the *key* is found inside *cached_hc*, corresponding ICs are computed. If not, the next hidden class is visited. Note that all the actual loading operations are guarded by the check of HI relationship. Based on the definition of HI relationship, STSC can enlarge the scope of cache-hitting hidden classes from the cached hidden classes to their descendant hidden classes in the same HI trees.

Especially, since the HI relationship has eliminated the name conflicts between local properties and vtable properties, the generated HI-based ICs only need to check if the loaded value is *uninit-marker* (the default value of object properties), and directly return the loaded value or *undefined*

without the need of traversing the prototype chain to search the property. If HI checking fails, the execution falls back to a runtime routine *Slow_Load()*. This routine will accomplish the loading operation and generate PIC to accelerate the load operation.

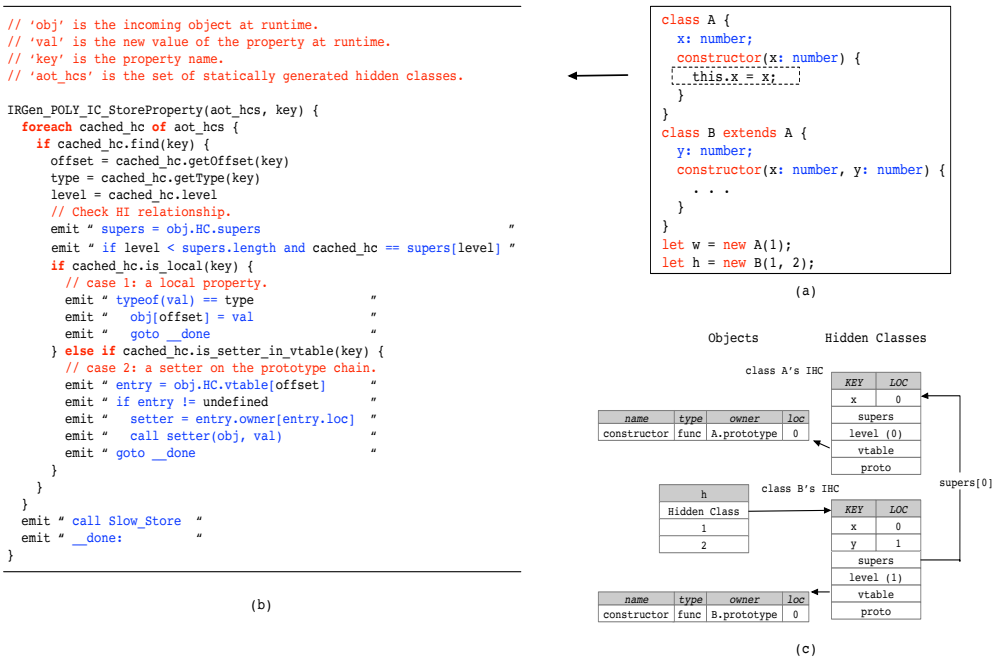


Fig. 9. (a) The example code in Figure 1 (b). (b) STSC's code generation algorithm for storing properties. (c) STSC creates two hidden classes as the IHCs of the two classes (A and B) and builds a HI-tree.

To understand the algorithm, we apply it to the program in Figure 8 (b). STSC first generates two hidden classes based on the two interfaces I1 and I2 as shown in Figure 8 (c). Then, STSC computes the locations of properties based on the hidden classes. When object *b* with the hidden class created from interface I2 is passed to function *foo*, no IC miss will be triggered since STSC can detect the HI relationship between the two hidden classes and directly load property *x* with location *0*.

Store Operations. The code generation algorithm of STSC for storing properties is shown in Figure 9 (b). Here we apply the generated HI-based ICs by this algorithm to the program in Figure 9 (a). When the object *h* in Figure 9 (a) is passed to the constructor of parent class A, an IC miss can be avoided as well. It is because STSC can detect the HI relationship between the cached hidden class (the IHC of class A) and the hidden class of *h*, which is the IHC of class B. After verifying the HI relationship, STSC can safely update the value of property *x*. If HI checking fails, the execution falls back to a runtime routine *Slow_Store()*. This routine will accomplish the storing operation and generate PIC to accelerate the store operation.

5.2.6 Generating Code. STSC provides two compilation modes to consume the generated HI trees and inline caches: *bytecode mode* and *native mode*. They aim to produce bytecode and optimized machine code, respectively. Here we briefly describe the implementations of these two modes.

Bytecode Mode. In this mode, STSC compiles TypeScript programs to bytecode. STSC first compiles typed JavaScript to ASTs. Then, it generates HI trees and function objects from the ASTs. Next, STSC computes HI-based inline caches and compiles the ASTs to bytecode. The generated inline

caches serve as meta data and are attached to the emitted bytecode. The final compilation results (e.g., function objects, hidden classes, inline caches, and bytecode) are packed into files. At runtime, the JIT compiler of STSC (a minorly modified version of V8's JIT compiler) can inline the AOTC generated hidden classes and HI-based inline caches into specialized code to reduce deoptimizations.

Native Mode. In this mode, STSC compiles TypeScript programs to optimized machine code. It first compiles TypeScript with the bytecode mode. Then, STSC translates the generated bytecode attached with the HI-based ICs into the form of V8's sea-of-node IR [Click and Paleczny 1995] to reuse the sophisticated optimization passes of V8's compilation framework. Next, STSC performs traditional optimizations, such as dead code elimination and constant propagation, on the generated IR and emit optimized machine code. Especially, due to the limitations of AOTC, some optimization passes, e.g., function context inlining and global object inlining, are disabled. Finally, all the compilation results (e.g., function objects, hidden classes, and machine code) are packed into files. In this way, the runtime of STSC can load these files and directly execute the deserialized optimized machine code. Note that the AOT compiler of STSC is a modified version of the JIT compiler of V8, but the details are out of the scope for this paper.

5.3 Runtime Phase

At the runtime phase, the runtime system of STSC loads the output of the compilation phase to executes the compiled TypeScript programs. Due to the unsoundness of static types of TypeScript and the dynamic features of TypeScript (e.g., dynamically adding and deleting properties), hidden class transitions are unavoidable at runtime. Therefore, another job of the runtime is to reduce IC misses by reconstructing or maintaining the AOTC generated HI relationships. In comparison with V8's runtime, one key enhancement of STSC's runtime is to reconstruct the HI relationship between the newly created hidden classes and the original hidden classes. In this way, runtime IC misses can be reduced. Another enhancement of STSC is to modify the implementations of various built-in querying and enumerating operations (e.g., *hasOwnProperty()* and *for-of*) by adding an extra check to filter the uninitialized properties, while searching or enumerating object properties. In this section, we focus on how to reconstruct HI relationships when dynamic operations take place.

5.3.1 Dynamic Operations. The static types may be violated at runtime due to the unsoundness of TypeScript's type system and the dynamic nature of TypeScript, so the cached class hierarchy information may be invalidated by dynamic operations. We focus on three typical operations, property additions, property deletions, and property assignments with changes of prototype chains. These dynamic operations have also been investigated in [Richards et al. 2010], which shows these operations, such as changing prototype objects and deleting object properties, are not infrequent. Figure 10 (a) shows the HI tree and prototype chain for the sample code with two classes A and B. Their IHCs, HC15 and HC16, are statically created by STSC and linked to form a HI tree, HC20 $\lhd$ HC21. The detailed structures of HC20 and HC21 are displayed in Figure 9 (c). The other six subgraphs display the three dynamic operations for normal objects or prototype objects.

5.3.2 Maintaining or Reconstructing HI Relationships. STSC translates these dynamic operations with corresponding runtime routines. In the runtime routines, STSC always tries to preserve the original HI relationships or reconstruct new ones. This optimization of reconstructing HI relationship is based on our observation: not all the dynamic operations violate the HI relationships. Therefore, the key design of the reconstruction mechanism is to build a HI relationship between the original hidden class and the new one created by the dynamic operations. In this way, the new hidden class can be linked to the HI tree of the original hidden class. As a result, runtime IC misses

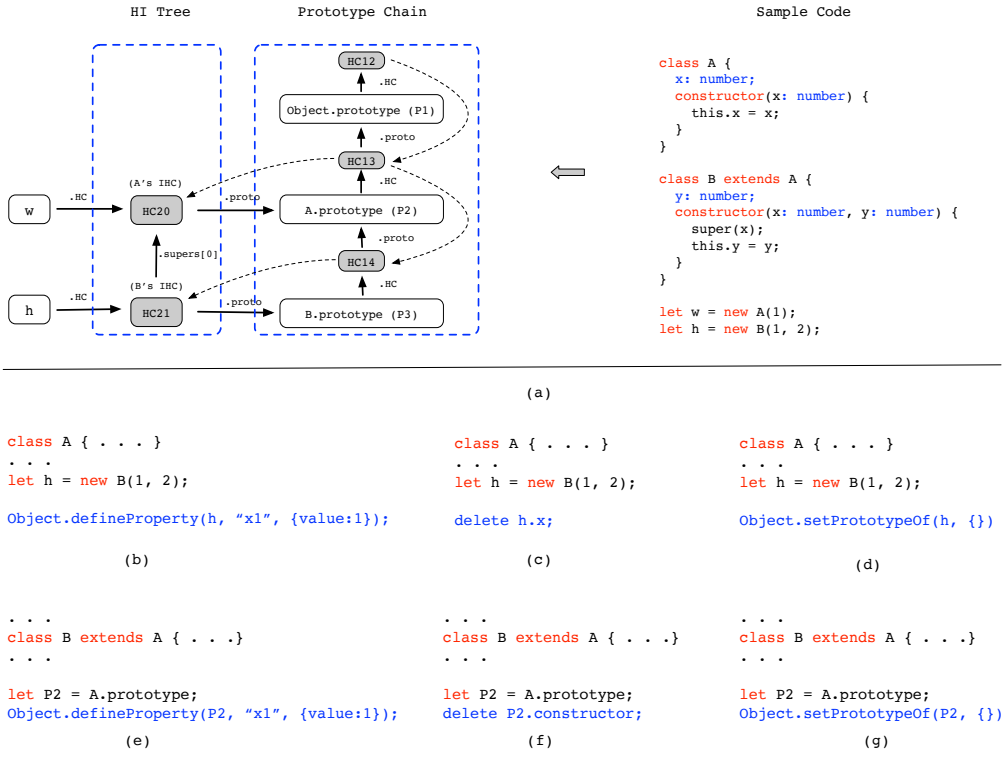


Fig. 10. (a) The right side is the sample program consisting of two classes A and B. The left side displays the corresponding HI tree and prototype chain. The HC20 and HC21 are the IHCs of the two classes. The HC12, HC13, and HC14 are the hidden classes of the three prototype objects P1, P2, and P3, respectively. The dashed lines indicate the back-references in the prototype chain. (b) Adding a new property *x1* to instance *h*. (c) Deleting a property *x* from instance *h*. (d) Changing the prototype object of instance *h*. (e) Adding a new property *x1* to prototype object P2. (f) Deleting a property constructor from prototype object P2. (g) Changing the prototype object of prototype object P2.

can be reduced. We list the impact of the three dynamic operations in Table 6. After identifying the impacts, we design a reconstruction mechanism to maintain HI relationships at runtime.

Table 6. How STSC handles the impacts of dynamic operations for HI relationship.

	Adding a new property	Deleting a present property	Changing the prototype chain
Normal object	Increase local properties and might violate the third condition of HI.	Keep HI relationship since STSC does not create new hidden classes.	Change vtable properties and might violate the third condition of HI
Prototype object	Change vtable properties and might violate the last three conditions of HI.	Keep HI relationship since STSC only needs to update the affected vtables.	Change vtable properties and might violate the last three conditions of HI.

To understand the reconstruction mechanism of STSC, we take the simple OO-style program in Figure 10 (a) as an example. Let's consider these six cases in Figure 10 (b-g). We first consider the dynamic operations for normal objects, such as the object *h* in Figure 10(a).

(1) For (b), when a new property *x1* is added to object *h*, the runtime routine *defineProperty()* is invoked and a new hidden class denoted as HC21* is created, similar to the transition

mechanism of V8. Then, STSC checks if a new HI relationship can be constructed between HC21 and HC21*. According to the definition of HI relationship, if the new property *x1* does not shadow any vtable property in the vtable of HC21, the HI relationship can be established. So, STSC links HC21* to HC21 by copying the *supers* of HC21 and itself to HC21*. In this way, HC21* won't cause any IC misses of the ICs that cache HC21.

(2) For (c), in current V8 design, deletion operations always trigger hidden class transitions, since the new hidden classes must exclude the deleted properties to ensure the correct semantics of property querying. Unlike the design of V8, STSC implements deletion operations by simply updating the value of the target properties to *uninit-marker* when the owner objects are normal objects. Based on the design of the uninitialized properties and HI, this simple algorithm of STSC implements and maintains the semantics of deletion operations.

(3) For (d), similar to V8, when the prototype object of the normal object h is updated, a new hidden class denoted as HC21* is created by STSC since the *proto* field in hidden classes is immutable. Then, STSC tries to reconstruct the HI relationship between HC21 and HC21*.

(a) If the prototype properties on the new prototype chain of HC21* do not shadow any local properties of HC21, STSC copies the local properties and vtable properties of HC21 to HC21*. For each vtable property *vp* in HC21*, STSC traverses the new prototype chain of HC21* for verifying. If a prototype property with the same name as *vp* is found and its type is consistent with *vp*'s type, the owner and location of *vp* in the vtable of HC21* is updated. Otherwise, the vtable entry of *vp* is updated to *undefined*. After updating the vtable of HC21*, STSC links HC21* to the HI tree of HC21 by adding HC21 to the *supers* of HC21*.

(b) On the other hand, if the prototype properties on the new prototype chain of HC21* shadow any local properties of HC21, STSC does not reconstruct the HI relationship for the two hidden classes HC21 and HC21*.

Next, we consider the dynamic operations for prototype objects P1, P2, and P3 in Figure 10 (a). Instead of directly triggering IC misses like V8 (as explained in Section 2.2), STSC tries to reconstruct HI relationships when these dynamic operations occur. Before performing the following HI reconstruction, STSC first collects the *affected hidden classes* by traversing the back-references on the prototype chain. For instance, the affected hidden classes of prototype P2 are {HC20, HC21}.

(1) For (e), if a new property *x1* is added to the prototype object P2, the hidden class HC13 of P2 will transit to a new hidden class denoted as HC13*. Then STSC visits all the affected hidden classes to verify HI relationship and update their vtables. For each affected hidden class, if there is any vtable property with the same type with the new property is shadowed, the owner and location of the vtable property is updated. On the other hand, if there is a shadowed vtable property with a different type or a shadowed local property on the affected hidden class, the HI relationship is violated. Next, the affected hidden classes are trimmed from the HI trees by clearing the contents of their *supers*.

(2) For (f), similar to the algorithm for (c), STSC updates the value of the deleted property constructor to *uninit-marker*. Furthermore, the affected hidden classes' vtables are updated as well. If the deleted prototype property shadows any vtable property of the affected hidden classes, STSC updates the owner and location of the vtable property. In this way, the original HI relationships are always reserved.

(3) For (g), When the prototype chain of P2 is updated. For each affected hidden class of P2, HC20 and HC21, STSC tries to reconstruct HI relationships using the algorithm for (d), since this operation can be regarded as changing the prototype chains of the affected hidden classes.

From the presented algorithms, we can see that by reconstructing HI relationships, STSC handles most of the dynamic operations and maintains HI relationships at runtime. Especially, unlike

the IC design of V8, changing prototype chains will not trigger IC misses or deoptimizations in STSC as long as the original HI relationships have not been broken or new HI relationships can be constructed. For the two corner cases of property shadowing between local properties and prototype properties, STSC fails to reconstruct HI relationships for the affected hidden classes. Fortunately, some popular frameworks like Vue.js [Vue.js 2019] and React.js [Facebook 2013] relied on overriding the callback methods of base components to develop applications instead of introducing the property shadowing between local properties and prototype properties. STSC's HI technique naturally supports this popular programming pattern.

6 DISCUSSION OF LIMITATIONS

Although STSC can work on the unsound static types of TypeScript, there is a limitation of our approach: the quality of the static types significantly impacts the performance of the generated specialized code and inline caches. If a TypeScript program is annotated with a lot of *any* type, the performance of the generated code by STSC will be degraded. Besides, shipping pre-compiled executable machine code is not possible for some browser applications. Therefore, STSC is suitable for native/hybrid applications or frameworks written with TypeScript. However, we believe that as more applications are written with TypeScript, STSC provides an alternative approach to exploit the static type information for TypeScript performance. To demonstrate the limitation and efficiency of STSC, Section 7 applies STSC to industry benchmarks and real-world applications.

7 EVALUATION

We evaluate the performance of STSC with two kinds of benchmarks: ported JavaScript programs and real-world TypeScript programs. We assemble benchmarks in this way as it represents the two popular usage patterns of TypeScript. One pattern of using TypeScript is to annotate existing JavaScript programs with static types and port them into TypeScript, while another pattern is to develop applications with TypeScript from scratch. We select the OO-style JavaScript benchmarks of the JetStream2 [WebKit 2019] benchmark suites and port them to TypeScript by manually adding static types for the variables and classes. Since STSC focuses on improving the performance of OO-style TypeScript programs, the selected benchmarks contain at least two classes or interfaces with inheritance relationship. For these benchmarks, we have excluded code loading, JSON parsing and regular expression related benchmarks since they are irrelevant to JavaScript's performance. Also, we have eliminated browser-only, Web Worker, and WebAssembly related benchmarks for apparent reasons. Furthermore, we have also removed very large benchmarks (larger than 30000 lines of code) due to the high cost of porting. For real-world TypeScript programs, we select a popular TypeScript application TSC (the official compiler of TypeScript v3.8 [Microsoft 2014]) and a popular UI component framework Vue.js [Vue.js 2019]. These two large-scale OO-style TypeScript programs are developed with TypeScript from scratch. We use TSC to compile itself and measure the compilation time for the TSC benchmark. To evaluate the framework Vue.js, we select its example applications (grid, svg, todomvc, and tree) that create various UI components by invoking the core APIs of Vue.js. For these UI applications, we only measure the execution time of TypeScript and exclude the rendering time.

All these benchmarks and applications are displayed in Table 7. The 'Lines of Code' row indicates the lines of the TypeScript code. The LoC of the four Vue.js applications include the LoC of Vue.js framework. The 'Manual Types' row indicates the numbers of the added static type annotations. Note that all the numbers of 'Manual Types' of real-world benchmarks are zero since we do not add any types to them. The 'Classes' and 'Interfaces' rows indicate the numbers of classes and interfaces in the benchmarks, respectively.

Table 7. The sizes and static types of benchmarks.

	Ported JavaScript							Real-world TypeScript				
Benchmarks	Unipoker	Richards	Deltablue	Raytrace	Babylon	Air	WSL	TSC	Todomvc	Tree	SVG	Grid
Lines of Code	371	415	586	744	5607	13925	14421	109951	23342	23348	23389	23310
Manual Types	52	79	150	147	1717	1753	2249	0	0	0	0	0
Classes	5	9	13	18	14	14	125	7	0	0	0	0
Interfaces	0	0	1	0	21	11	19	778	105	107	105	106

Since STSC is implemented on the V8-7.7.299.4 version, we choose this version of V8 as our baseline. We transpile the TypeScript programs to typed JavaScript programs and execute them with STSC and baseline V8. To demonstrate the effect of the HI technique of STSC, we design the following two experiments:

(1) To evaluate the performance of STSC's bytecode mode, we use two configurations B_b and S_b. The baseline (B_b) configuration is the original V8, while the configuration S_b is the bytecode mode of STSC. B_b pre-compiles all the benchmark programs into bytecode with the *CodeCaching* [Google 2015] technique of V8. Then, B_b executes the benchmarks after loading the CodeCaching files. Similarly, S_b first compiles the benchmark programs with bytecode mode and then loads the output before executing the benchmarks. In this way, the overheads of bytecode compilation can be excluded from the executions. The purpose of the design is to focus on the impact of the HI technique on execution performance.

(2) To evaluate the performance of STSC's native mode, since V8 does not provide the capability of statically compiling plain JavaScript programs to optimized machine code, we use two configurations (B_n and S_n) of STSC to evaluate the impact of the HI technique on the generated code. The baseline configuration (B_n) is the native mode of STSC disabling HI technique, which can be achieved by replacing the HI-based ICs (described in Section 5.2.5) with conventional ICs. The configuration S_n is the native mode of STSC with HI-based ICs. Both configurations S_n and B_n firstly compile the JavaScript programs to AOT files and then load them to execute. At runtime, the JIT compiler of STSC is disabled for both configurations to focus on the performance of the AOTC generated code.

The experiments are performed on a quad-core Intel Core i7-4770HQ 2.2GHz OSX version 10.12.6 (64-bits) machine with 16GB RAM and 250GB FLASH. All the benchmarks and applications are executed on one of the CPUs with Turbo Boost disabled to avoid instability in the results. All measurements are repeated ten times and reported numbers are the average of the runs. We use the API *mach_absolute_time()* on the macOS platform to measure the execution time. The API can achieve a high resolution in microseconds.

7.1 The Performance of STSC's Bytecode Mode

Results reported in Figure 11 focus on the average performance for B_b and S_b. The average performance are normalized to B_b. The numbers in parentheses are the iterations. We use the original iteration numbers of benchmark suite JetStream2. The absolute times are shown on the right of the B_b bars. We can see that S_b can significantly reduce the average execution times of the six benchmarks by 8.75% (WSL), 15.17% (TSC), 17.74% (Grid), 15.56% (SVG), 20.03% (Tree) and 10.21% (Todomvc). For other benchmarks, S_b does not show notable improvement due to the small numbers of interfaces and classes. To investigate the impact of the HI technique, we profile the execution using V8's sophisticated profiling tools via enabling the option '—enable-tracing'. Based on the profiling data, we assess the performance improvement from four dimensions: hidden class allocations, IC misses, JIT compilations and deoptimizations. Figure 12 displays the numbers of these

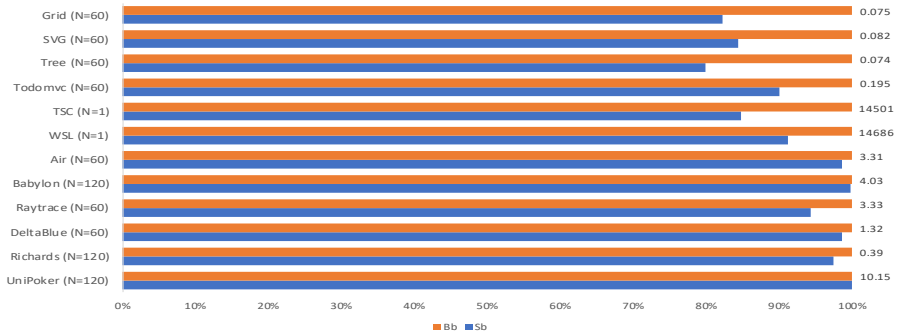


Fig. 11. The average performance of configurations B_b and S_b. The number N in parentheses indicates the number of the iteration. On the right of B_b bars, we show the absolute time in milliseconds.

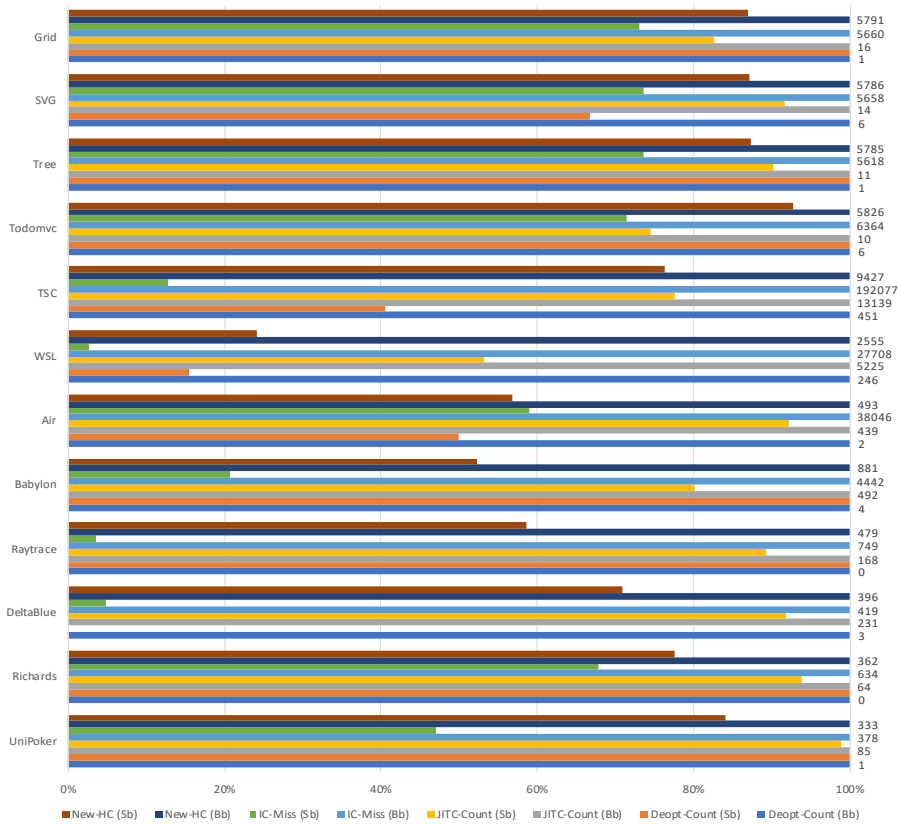


Fig. 12. The numbers of creating hidden classes, IC misses, JIT compilations and deoptimizations in configurations B_b and S_b. On the right of the B_b bars, we show the absolute numbers of creating hidden classes, IC misses, JIT compilations and deoptimizations.

dimensions in configurations B_b and S_b. First, we notice that S_b reduces about 15%~75% hidden class transitions relative to baseline B_b, which demonstrates the effect of statically generating IHCs in STSC. In addition, we can see that the numbers of IC misses are significantly reduced, e.g., 30%(Todomvc), 98%(WSL), and 86%(TSC). Besides, configuration S_b can significantly reduce the number of deoptimizations by 84% (WSL) and 59%(TSC). It reduces the times spent on interpretation execution and JIT compilation. On average, the S_b reduces 58% IC misses and 39% deoptimizations in comparison with B_b. The improvements of these two dimensions demonstrate the efficiency of our HI design, especially for the large-scale OO-style benchmarks. With the decreases of deoptimizations and IC misses, the numbers of JIT compilations are decreased by 46%(WSL) and 22%(TSC).

7.2 The Performance of STSC's Native Mode

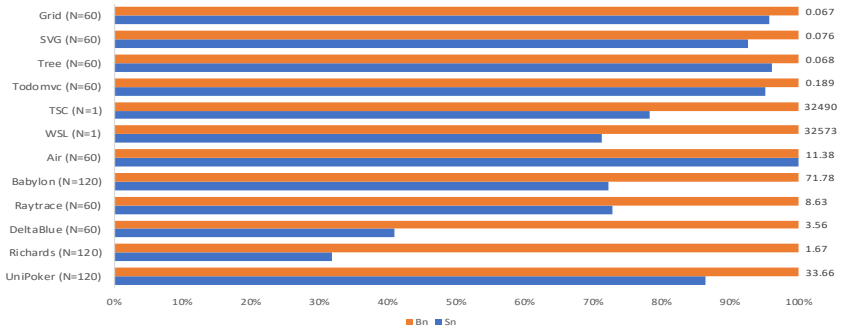


Fig. 13. The average performance of configurations B_n and S_n. On the right of the B_n bars, we show that absolute time in milliseconds.

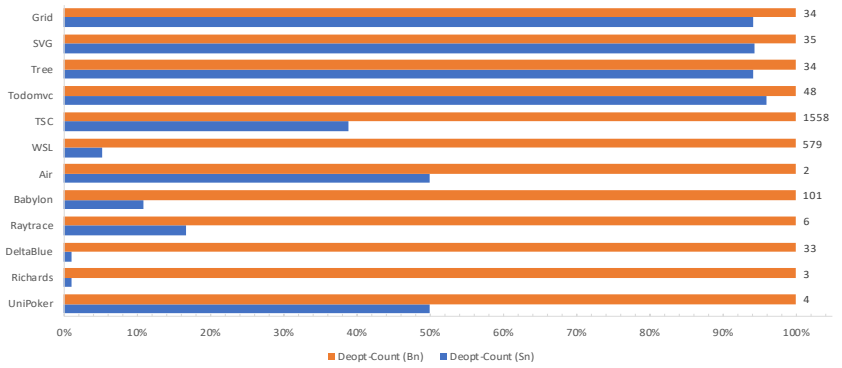


Fig. 14. The numbers of deoptimizations for configurations B_n and S_n. On the right of the B_n bars, we show the absolute numbers of creating hidden classes, IC misses, JIT compilations, and deoptimizations.

Figure 13 reports the average performance of statically generated machine code of configurations B_n and S_n. Note that B_n and S_n disable the JIT compilation of STSC during the executions to measure the quality of the AOTC generated code. From Figure 13, we can see that S_n significantly

reduces almost all the execution times by 6% ∼ 67% except for the benchmark Air. We assess the improvement by profiling the numbers of deoptimizations. Figure 14 displays the numbers of deoptimizations for both configurations. We can see that S_n can reduces the deoptimizations triggered by IC misses by 5% ∼ 94% for benchmarks. It demonstrates the efficiency of our HI design in the dimension of deoptimization. We also note that although S_n eliminates half of the deoptimizations of the benchmark Air, the execution time is not decreased due to the small count (2) of deoptimizations.

Comparison with Other AOT Techniques. STSC uses a modified version of the JIT compiler (Turbofan) of V8 in native mode to statically generate optimized machine code (described in Section 5.2.6). Although V8 does not provide the capability of statically compiling plain JavaScript programs to optimized machine code, we are interested in the comparison between configuration S_n and other AOT techniques. We compare STSC (with configuration S_n) with Hopc compiler [Serrano 2018] and list the results in Table 8. Based on the numbers in Table 8, we can see STSC outperforms Hopc for most benchmarks (e.g., Deltablue and Richard) when normalized to their baseline V8 versions. However, since most benchmarks except the Deltablue and Richard focus on numerical calculation and bit logic operations, STSC with configuration S_n falls behind the baseline V8-7.7. It is because the *number* type of TypeScript doesn't distinguish integer types from float types, which is essential for generating efficient machine code for numerical and logic benchmarks. The current implementation of the native mode of STSC employs boxed numbers to represent the values annotated with *number* type when statically generating machine code. Since enhancing the static types of TypeScript is out of the scope of this paper, we leave it as future work.

Table 8. The performance of STSC (configuration S_n) and Hopc relative to baseline V8-7.7 and V8-6.2, respectively. We normalize the execution times of STSC to its baseline V8-7.7, while the reported times of Hopc [Serrano 2018] are normalized to V8-6.2.

Benchmarks	**Crypto-md5**	**Deltablue**	**Richard**	**Crypto-aes**	**Base64**	**Crypto**	**Splay**
STSC / v8-7.7	5.91	1.14	1.24	2.32	1.07	6.01	1.06
Hopc / v8-6.2	6.51	6.81	3.34	2.34	0.59	7.31	1.17

7.3 The Warmup Performance of STSC

In addition to average performance, we are also interested in the warmup performance of STSC. To evaluate warmup performance, we increase the iteration number N (up to 200 or 50) and report the average execution times up to each iteration for the four configurations in Figure 15. For instance, the average execution time of iteration ten means the average execution time of the first ten iterations. As the increase of iteration numbers, the average execution times of all the benchmarks are reduced. For some short-time benchmarks, excluding benchmarks WSL, TSC, and Babylon, configuration S_n achieves speedup 1.5x ∼ 6.1x for startup performance (the execution time of the first iteration) relative to configuration B_b. Comparing configurations B_b and S_b, we observe that S_b does not significantly improve the warmup performance of the small size benchmarks. However, for the large benchmarks, WSL, TSC, and the four Vue.js applications, S_b can significantly improve the warmup performance especially for the small iterations less than fifty.

7.4 The Comparison between STSC's Native Mode and V8's JITC

We are also interested in the performance comparison between configurations S_n and B_b. We choose the average execution times of the max iterations in Figure 15 to evaluate the performance for the two configurations. On average, S_n achieves about 54.2% performance of B_b, which means the performance of the native mode of STSC achieves half of the performance of V8's JIT compiler.

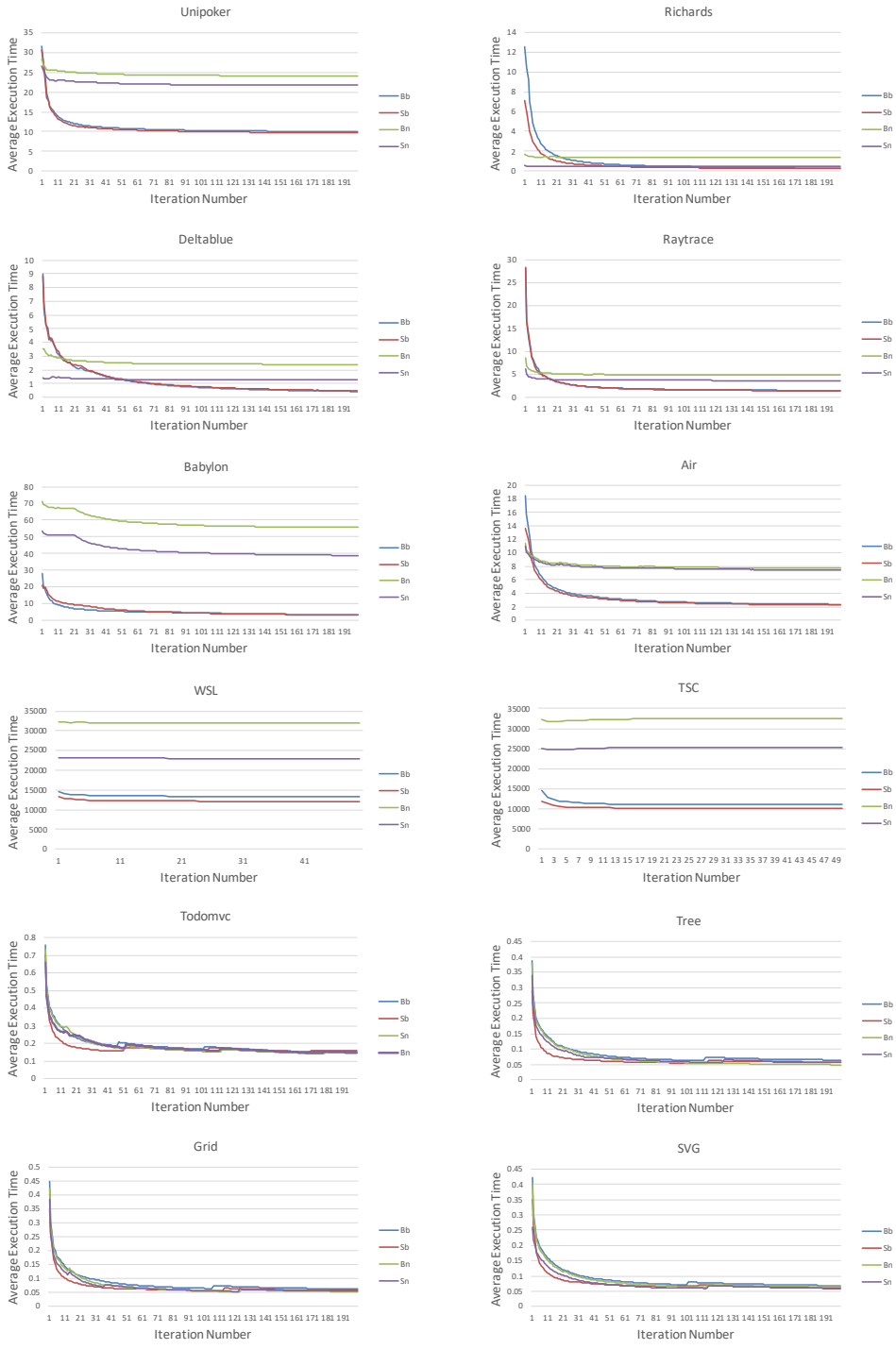


Fig. 15. The warmup performance of the four configurations. The vertical axis shows the average times of iterations, while the horizontal axis displays the iteration numbers.

7.5 The Statistics of Reconstructing HI Relationships

Table 9. The numbers of the total reconstructions and the successful reconstructions for HI relationships at runtime.

	Unipoker	Deltablue	Richards	TSC	Raytrace	Babylon	WSL	Air	Todomvc	Tree	Grid	SVG
Reconstruction	0	0	0	768	0	0	0	0	0	0	0	0
Success	0	0	0	768	0	0	0	0	0	0	0	0

To evaluate the reconstruction mechanism of HI relationships (section 5.3.2), we count the numbers of reconstructing HI relationships. We select the same experiment in section 7.3 with configuration S_b and record the numbers of reconstructions. Table 9 lists the numbers of the total reconstructions and the successful reconstructions. We notice that STSC can successfully reconstruct HI relationships at runtime for the real-world benchmark TSC. Since the benchmark TSC heavily uses type assertions and dynamically defines properties on the casted objects, it triggers hundreds of HI reconstructions at runtime. For the ported JavaScript programs, we provide precise type annotations to the variables and objects, e.g., all the properties of classes are annotated in their declarations, so HI reconstructions do not take place. In the Vue.js applications, their pre-existing static types are precise and no such operations like changing prototype chains are performed, so HI reconstructions are also eliminated at runtime.

7.6 The Sizes and Loading Times of AOT Files

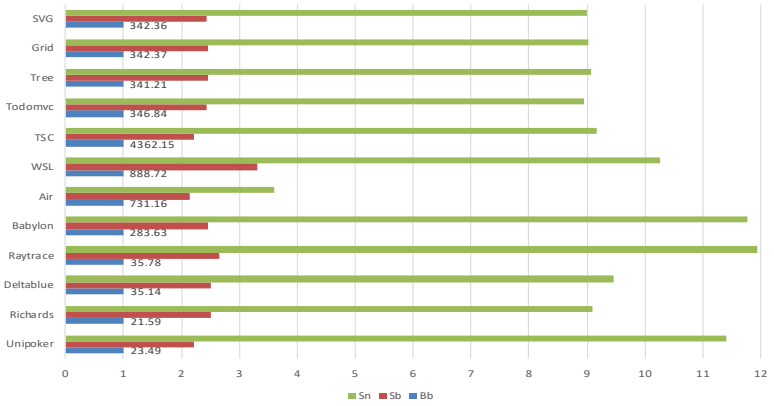


Fig. 16. The sizes (KB) of the CodeCaching files of configuration B_b, and AOT files of configurations S_b and S_n. The numbers of S_b and S_n are normalized to B_b. On the right of the B_b bars, we show the absolute sizes (KB) of the CodeCaching files of configuration B_b

We are also curious about the sizes and loading times of AOT files. Figure 16 reports the sizes of AOT files generated with configurations S_b and S_n. For configuration S_b, we can see that the AOT files increase by 2.1x~2.6x in comparison with B_b. For configuration S_n, the sizes increase by 3.6x~11.9x due to the generated specialized code. In Figure 17, we report the loading times of configurations S_b and S_n, which are normalized to the loading times of the baseline configuration B_b. The average loading times of S_b and S_n are 2.46x and 6.79x slower than B_b. From the statistics in Figure 16, we can see the generated AOT files are relatively large, since all configurations compile

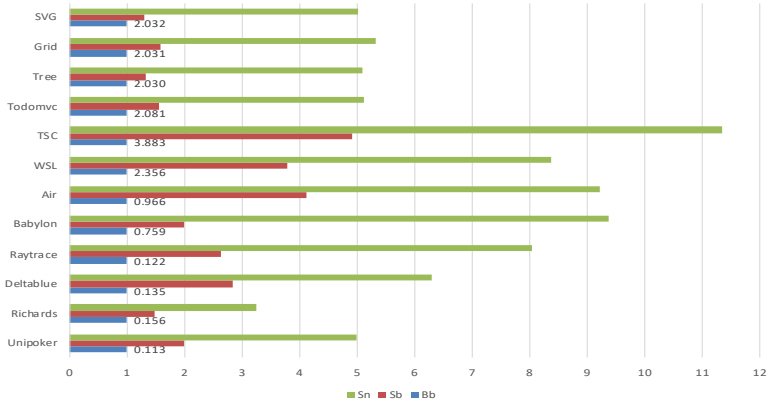


Fig. 17. The loading times of S_b and S_n normalized to B_b. On the right of the B_b bars, we show the absolute loading time (in millisecond) of configuration B_b.

all the functions. Therefore, we can decrease the sizes by selecting the hot functions via profiling information and it can be future work.

8 RELATED WORK

Many efforts have been made to improve the performance of JavaScript. Since TypeScript is a superset of JavaScript and can be transpiled to plain JavaScript, similar techniques will work for TypeScript. Hopc [Serrano 2018; Serrano and Feeley 2019] is similar to our work in terms of supporting unrestricted JavaScript. It implements AOTC for JavaScript through static analysis and generating versioning-based [Castanos et al. 2012] specialized code based on the unsound type information from type inference. Hopc generates slow paths for each dynamic operation to handle type violations. This approach relies on a profiling run to collect hidden classes, while our STSC statically generates HI trees and HI-based ICs leveraging the static type information. Hopc also proposes a vtable-like technique to accelerate the access of local properties by recording all the possible property caches in all the property access sites. Unlike STSC's vtable, this approach cannot be applied to accelerate the access of the prototype properties, and it requires more memory resources since the sizes of Hopc's vtables increase with the number of the property access sites in programs.

Mori [Clifford et al. 2015] proposes a profiling-based technique to optimize object representations for arrays. Mori focuses on profiling the transitions of the hidden classes of array objects due to the changing of the kinds of array elements. However, their technique does not mention how to avoid the transitions caused by the property additions or deletions for normal objects rather than array objects. Besides, they do not focus on how to reduce the IC misses or deoptimizations due to the polymorphic behaviors in OO-style TypeScript programs. On the other hand, since our HI technique and their allocation-site technique are orthogonal, both techniques can be combined to reduce IC misses or deoptimizations.

On the other hand, Park et al. [Park et al. 2017] focus on saving the compiled optimized code during executions. Their system saves optimized code in previous runs and drops the embedded hidden classes. At the next execution, the runtime re-patches the deserialized optimized code with the hidden classes created at runtime. However, their approach does not mention how to generate efficient ICs for OO-style JavaScript / TypeScript programs, where the highly polymorphic

behaviors degrade the performance of conventional ICs. Choi et al [Choi et al. 2019] propose a reusable IC design (RIC) to improve JavaScript performance, which saves the IC information in an execution in a context-independent way. Comparing with RIC, our HI technique focuses on leveraging class hierarchy information to statically generate enhanced hidden classes and ICs, which can't be generated and saved from the profiling runs of the RIC design.

StrongScript [Richards et al. 2015] extends TypeScript with concrete types, which are used to specify the layouts of objects. Their approach requires programmers to identify the type-safe parts of programs, so the compiler can safely remove associated dynamic type checks to improve the performance. Comparing with StrongScript, STSC does not require the soundness of static types and maintains the semantics compatibility for JavaScript, which are fundamental differences between them. STS [Ball et al. 2019] compiles a subset of TypeScript to efficient machine code. It excludes many of the dynamic features of JavaScript (such as eval routine and prototype-based inheritance), while STSC supports all the dynamic features of TypeScript.

9 CONCLUSIONS AND FUTURE WORK

Our HI technique presents an IC design for improving TypeScript performance. It is a practical approach and can be built on established infrastructure. The HI technique provides an alternative approach to use the unsound static types of TypeScript without breaking the language. We implement the HI design in STSC built on a commercial JavaScript VM and demonstrate performance improvements on industrial benchmarks and applications. Besides, as mentioned in this paper, the quality of static types significantly impacts the performance of STSC. One direction of future on STSC is to introduce a more sophisticated and powerful type inference. Another direction is to integrate profiling runs to improve the quality of the static types of TypeScript.

ACKNOWLEDGMENTS

We would like to thank the reviewers for valuable feedback. Besides, we also thank KangHao Lv for his extensive suggestions.

A APPENDIX

Recalling static types are encoded into JSON strings (TA), and each function or class is given a *tid* in the form of comments. Figure 18 illustrates the structures of TA. The TA in the typed JavaScript records the basic types: *any*, *number*, *void*, *string*, and so on. Note that the contents of class A's TA includes *supers*, *fields*, and *methods*, which are corresponding to the type annotations in the original TypeScript program. In addition, we can note that the associated comment of variable A encodes its *tid* 8. In this way, each variable can be associated with its static type.

```
/* "TA": [ "any", "number", "void", "string", . . .
          { "cname": "B",
            "supers": { "A": 8},
            "fields": { "y": 1 }
            "methods": { . . . }
          }
          . . .
        ]
*/
class /*<@8>*/A {
  constructor(/*<@1>*/x) { . . . }
  . . .
}
class /*<@9>*/B extends A { . . . }
```

Fig. 18. The structures of TA.

REFERENCES

Angular2. 2017. Google Angular JS Framework. https://developers.google.com/v8/.

Apple. 2018. JavaScriptCore. https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore.

Artoul. 2015. Javascript Hidden Classes and Inline Caching in V8. http://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html..

Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: An Implementation of a Static Compiler for the TypeScript Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

Bevenius. 2018. Learning Google V8. https://github.com/danbev/learning-v8..

Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. 2012. On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 195–212.

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 49–70.

Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 889–901.

Cliff Click and Michael Paleczny. 1995. A Simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*. ACM, New York, NY, USA, 35–49.

John Rose Cliff Click. 2002. Fast subtype checking in the HotSpot JVM. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java (JGI 2002)*. ACM, New York, NY, USA, 96–107.

Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117.

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302.

Egret-3d. 2017. Egret 3D Game Engine. https://github.com/egret-labs/egret-core.

Facebook. 2013. React.js Framework. https://reactjs.org/.

Ben Frederickson. 2015. Ranking. https://www.benfrederickson.com/ranking-programming-languages-by-github-users/.

Google. 2015. Code Caching. https://v8.dev/blog/code-caching.

Google-V8. 2019. V8 JavaScript Engine. https://developers.google.com/v8/.

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38.

IDE-VSCode. 2017. Microsoft Visual Studio Code - Open Source. https://github.com/Microsoft/vscode.

Microsoft. 2014. TypeScript Specification. http://www.typescriptlang.org/.

Microsoft. 2018. ChakraCore. https://github.com/Microsoft/ChakraCore.

NativeScript. 2017. NativeScript: A Framework of Native Mobile Applications. https://github.com/nativescript.

HyukWoo Park, SungKook Kim, and Soo-Mook Moon. 2017. Advanced Ahead-of-time Compilation for Javascript Engine: Work-in-progress. In *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion (CASES '17)*. ACM, New York, NY, USA, Article 16, 2 pages.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1806596.1806598

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

Manuel Serrano. 2018. JavaScript AOT Compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)*. ACM, New York, NY, USA, 50–63.

Manuel Serrano and Marc Feeley. 2019. Property Caches Revisited. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. ACM, New York, NY, USA, 99–110.

Stackoverflow. 2017. Programming Languages. https://insights.stackoverflow.com/survey/2017technology.

Superpowers. 2018. Superpowers. https://github.com/superpowers/superpowers-core.
Tiobe. 2020. Programming Languages. https://www.tiobe.com/tiobe-index/.
Vue.js. 2019. Vue.js Framework. https://github.com/vuejs/vue-next.
WebKit. 2019. JetStream2 Benchmarks. https://github.com/WebKit/webkit/tree/master/PerformanceTests/JetStream2.