On Policy Decisions of Polymorphic Inline Caches in Dynamically-Typed Language Implementations

by

Nathan Henderson

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Dynamically-typed languages running on Virtual Machines (VMs) are commonly used, but the lack of explicit type information poses a challenge to producing efficient code. In general, without type annotations, it is impossible to statically infer an object's type to determine which methods to invoke or how properties are accessed. Inline caches (ICs) are a widely adopted technique to improve the performance of dynamically-typed languages. ICs store machine code stubs at the bytecode level to enable fast-path execution for previously seen types. However, highly polymorphic sites require a large number of fast paths, leading to more frequent code generation and a higher runtime cost to select the correct fast path for an incoming type. Therefore, implementations often set a limit on the number of IC fast paths for a bytecode. Once this limit is reached, type-specialized fast paths are forgotten and instead, the IC executes a type-generic routine.

The central goal of this thesis is to investigate and evaluate alternative techniques to handle high degrees of polymorphism in operations that use inline caches. This thesis introduces *Stub Folding*, a technique that increases the efficiency of highly polymorphic ICs. Stub Folding allows certain ICs to retain type-specialized fast paths that would otherwise be lost, enabling higher code coverage for compiler optimizations and accelerating lower execution tiers. An implementation of Stub Folding in the SpiderMonkey JavaScript engine achieves up to 25% improvement on complex applications within the JetStream 2.1 benchmark suite compared to SpiderMonkey's previous approach. This

thesis also explores techniques inspired by hardware caching policies, namely Least Recently Used (LRU) and Least Frequently Used (LFU) replacement policies. An evaluation indicates that LRU and LFU policies accelerate some programs but do not reliably increase program efficiency across a range of benchmarks.

Preface

Chapter 4 of this thesis is published as J. de Mooij, M. Gaudet, I. Ireland, N. Henderson, and J. N. Amaral "CacheIR: The Benefits of a Structured Representation for Inline Caches". J. de Mooij designed and implemented the initial version of CacheIR and reviewed the manuscript. J. de Mooij, I. Ireland, and M. Gaudet have worked on improving the CacheIR system in their capacity as employees at Mozilla. I. Ireland contributed to the writing of the manuscript. M. Gaudet and I wrote a majority of the manuscript. Moreover, I designed and ran the experiments. J. N. Amaral contributed in his capacity as a supervisor and helped to review the manuscript.

Chapter 5 of this thesis published as N. Henderson, I. Ireland, M. Gaudet, J. P. L. de Carvalho, and J. N. Amaral "Stub Folding: Retaining Type Specialization to Increase the Efficiency of Highly Polymorphic Inline Caches" at the *33rd Annual International Conference on Computer Science and Software Engineering* in Las Vegas, NV on September 2023 where it won the *Best Paper* award. J. N. Amaral and J. P. L. de Carvalho contributed in a supervisory capacity; setting up industry collaboration, helping guide the project, and editing the final manuscript. I. Ireland and M. Gaudet were collaborators working at Mozilla. Both of them contributed greatly to exploring ideas, helping with implementation and evaluation, and reviewing the manuscript. I was responsible for implementing and extending the Stub Folding idea, designing and running experiments, conducting a literature review, and writing the manuscript.

Валерию

If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

– Ken Thompson, Reflections on Trusting Trust, 1984.

Acknowledgements

I want to give thanks to my supervisor José Nelson Amaral for helping me grow as a researcher, communicator, and person. He provided priceless advice on writing, communicating, the value of hard work, and perseverance; for that, I am forever grateful. I want to thank Iain Ireland and Matthew Gaudet for the many hours of effort put in to help me through this research including idea generation, writing, general advice, and sharing software engineering expertise. I would like to give thanks to the group of students and post-docs that I was able to work with and become friends with throughout my degree. This includes Caio and João for being great friends and collaborating on a successful research project; Dhanraj and Patrick for friendship and providing passionate discussions about anything; Rouzbeh for the great chats we had over lunch and the lessons in Persian history. I would like to give thanks to my fiancée, Valery, and our canine family, Pippa, Vanya, Misha, and Richard for being such an important part of my life and promoting daily walks even when I did not feel like it.

I would like to thank the University of Alberta for providing funding and an atmosphere filled with fantastic researchers; being surrounded by such a supportive and driven community was very helpful to maintain motivation.

Contents

List of Tables

List of Figures

Glossary

Inline Cache (IC)

Intermediate Representation (IR)

JavaScript (JS)

JavaScriptCore (JSC)
: JavaScript engine used in the Apple Safari web browser.

Just-In-Time (JIT)

Object-Oriented Programming (OOP)

SpiderMonkey (SM)
: JavaScript engine used in the Mozilla Firefox web browser.

Static Single Assignment (SSA)

V8 (V8)
: JavaScript engine used in Google Chrome, Node.js, and any Chromium based the web browser.

Virtual Machine (VM)

Chapter 1

Introduction

Compared to statically-typed languages such as C, C++, and Rust, dynamically-typed languages like JavaScript and Python provide greater ease of use and flexibility, leading to decreased development times and simpler algorithmic development [22], [32]. However, these benefits are accompanied by a higher runtime cost to appropriately handle types dynamically [9], [32]. Thus, dynamically-typed language implementations have long sought to achieve the higher performance that is achieved for statically-typed languages while maintaining ease-of-use and expressiveness [3], [6], [7], [9], [32]. Dynamic compilation techniques enable language Virtual Machines (VMs) to optimize and compile code at runtime. One of the most important of these techniques for dynamically-typed languages is inline caching. Inline caches (ICs) accelerate execution at a bytecode level by providing native-code fast execution paths specialized to observed bytecode-operand types [5]–[7], [9], [15]. When Just-in-Time (JIT) compiling functions, if only a single type is observed at a given bytecode operation — a property load, for example — the fast path stored in the IC is inlined into the compiled function, providing very efficient execution for hot, type-stable, code paths. However, polymorphism is a difficult problem for ICs to handle. A bytecode with many observed types requires a large number of fast paths, leading to more frequent code generation and preventing the compiler from inlining the fast paths into optimized functions.

Dynamically-typed language implementations typically deal with polymorphism by setting a low limit on the number of type-specialized fast paths

stored in an IC. This limit is often arbitrarily chosen, and, when it is exceeded, all the type information accumulated for that IC is lost. For example, in SpiderMonkey [30], ICs have a maximum of six type-specialized fast paths. If a property-load operation that has seen six types of objects is given a new type of object, the VM removes the fast paths and instead selects a slower, generic VM routine to compute the result of the bytecode.

The central goal of this thesis is to investigate and evaluate alternative techniques to handle high degrees of polymorphism in operations that use inline caches. The main technique, *Stub Folding*, is motivated by the observation that, in some cases, the fast paths in highly polymorphic ICs share the same code, only differing in the data that they operate on. For highly polymorphic operations, Stub Folding uses code analysis to determine if an IC can retain type-specialized fast-path information by combining all fast paths into one instruction stream and thereby increasing the likelihood of fast-path inlining. Two other techniques investigate the suitability of applying cache replacement policies, such as Least Recently Used to reorder IC fast paths and Least Frequently Used to remove fast paths that are used infrequently.

This thesis' main contributions are presented in Chapter 5 and include:

- Stub Folding, a novel approach to handling polymorphic inline caches that consolidates type-specialized fast paths into a unified instruction stream and improves runtime performance by facilitating efficient IC code inlining.
- A Stub Folding implementation in the SpiderMonkey (SM) JavaScript engine and an evaluation compared to SM's previous approach.
- An evaluation of the viability of cache replacement policies for reordering and eliminating underutilized fast paths in polymorphic inline caches.

Additionally, Chapter 2 of the thesis provides a detailed overview of the Inline Caching schemes in modern dynamically-typed language implementations with a focus on the three main JavaScript engines in SpiderMonkey [30], JavaScriptCore [17], and V8 [33]. Chapter 4 further contextualizes SpiderMonkey's existing approach to inline caching by discussing CacheIR, an Intermediate Representation (IR) specialized for inline caches, and evaluating WarpBuilder, the type specializing tiered execution architecture enabled by CacheIR's design.

Chapter 2

Dynamically-Typed Languages, Speculative Compilation, and Polymorphism

This chapter discusses dynamically-typed languages and their type systems, the challenges posed by polymorphic operations, and how dynamically-typed language virtual machines use speculative compilation techniques to accelerate program execution.

2.1 Dynamic Typing

Languages[1] like SELF, Smalltalk-80, JS, and Python support many dynamic features such as modifying objects at run time by adding or deleting properties and evaluating a string variable as code. In many cases, allowing such dynamism means that static types are not known ahead of time before program execution. Simply, dynamic typing is the process by which a language virtual machine enforces type safety at run time rather than through ahead-of-time type checks.

To illustrate this concept, Figure 2.1 shows a trivial snippet of Rust code contrasted with a snippet of JS code that updates a `score` field [2] on an object.

Rust is statically typed. Listing (a) in Figure 2.1 shows how much information the Rust compiler has access to and can statically enforce at compile time. From the `update_score` function signature and body, the Rust compiler

[1] In the context of this thesis, *languages*, also refers to their implementation, not solely their specification.

[2] In JavaScript fields are called *properties*.

Static Typing: `Rust`

```
1 fn update-score(o: &mut Obj, v: i32) -> i32 {
2   o.score = o.score + v;
3   o.score
4 }
```

(a)

Dynamic Typing: `JavaScript`

```
1 function updateScore(o, v) {
2   o.score = o.score + v;
3   return o.score;
4 }
```

(b)

Figure 2.1: Static vs. dynamic typing.

observes the following: (a) `o` must be of type `&mut Obj`; (b) `v` must be an integer of type `i32`; (c) `score` must be stored at a consistent offset from the base of `o`'s memory and be of type `i32`; and (d) the function must return a value of type `i32`. The Rust compiler enforces these invariants at compile time, generating compile-time errors if any of them are violated, and rejecting the code. For example, raising an error if a `&mut String`-typed variable is passed instead of a `&mut Obj`-typed variable. By enforcing these invariants, the compiler generates performant code without the need for runtime type checks.

In Listing (b), the JavaScript source code looks similar to the Rust source code but there are a few noteworthy details: First, the types of `o` and `v` are unknown; Second, the type of `o.score` is unknown and could anything including a method, a `null` variable, or `undefined`; Third, since `o`'s type is unknown, the offset at which to access `score` from `o` is unknown and need not be consistent between types. For these reasons, dynamically-typed languages are often

initially interpreted to handle a diverse set of incoming cases at runtime. In Listing (b), though it may not be semantically what the programmer meant to do, this function could legally receive a `String`-typed variable for `o` and a `Floating-point`-typed variable for `v`. This action is well-defined in JS and produces a concatenation of `String`s, first converting `v` to a `String` and then appending it to the value of `o.score`. Since the objects themselves and their properties are also dynamic, storing an `Integer` to `o.score` at one point in the program, then storing a `String` at another point is completely valid.

The properties of languages that use dynamic typing make it difficult to generate efficient code for functions like `updateScore` without observing the flow of types at runtime; a compiler no longer has a set of invariants that will not be violated like in statically-typed languages.

Section 2.2 discusses how speculative compilation is used to infer types for dynamically-typed languages to produce efficient specialized code.

2.1.1 Prototype-Based Object Models

Though generally elided from programmers' mental models, dynamically-typed languages still need an extensively defined internal type system and object model. Many dynamically-typed language implementations for object-oriented languages have adopted a prototype-based object model similar to SELF. In prototype-based languages, the object and inheritance model relies on creating objects by cloning and extending other objects as opposed to constructing objects through class definitions as is common in other languages supporting Object-Oriented Programming (OOP) like Java and C#. Terminology varies between implementations but the idea remains the same. Each object has a property slot that holds a pointer to a prototype object. When creating an object, the prototype property slot is filled in with the object used for instantiation. Since prototypes are themselves objects, each one has its own prototype. The links between an object and its prototypes define a *prototype chain*.

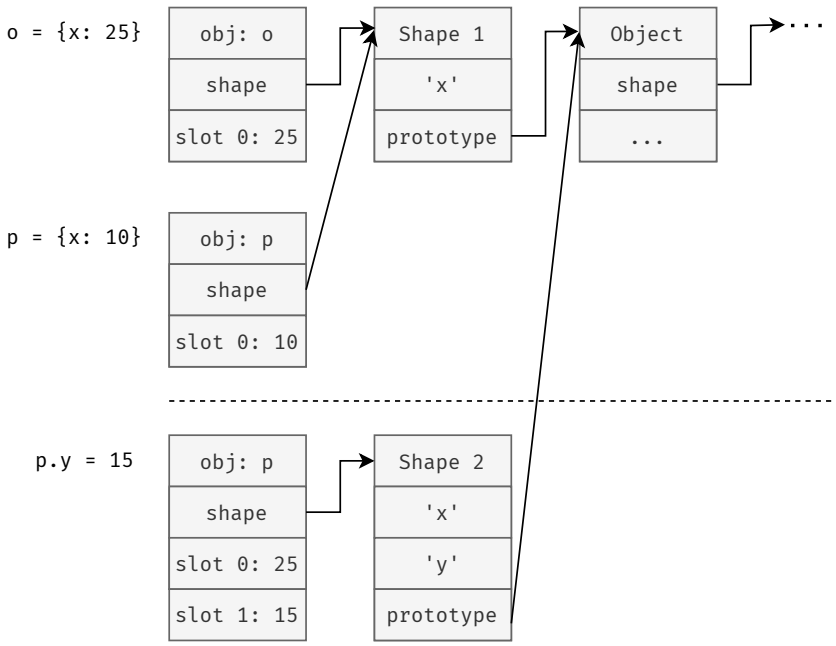

Figure 2.2: Example relationships between objects and shapes.

Type Systems via Object Shapes

In prototype-based languages, objects have: (a) a *shape* that describes the layout of the object in memory; and (b) *slots* that hold actual values. The shape of an object maps property-name strings to the corresponding slots where the properties can be found within the object. Objects that have the same mapping share the same shape. Shapes are immutable, meaning when a property is added to an object, a new shape must be associated with the object. If an existing shape matches the new mapping, it is associated with the object; otherwise, a new shape representing the correct set and order of properties is created and associated with the object. Because shapes are immutable, objects with the same shape can be handled similarly and operate under the same assumptions.

Figure 2.2 contextualizes the relationship between objects and shapes. Initially, object `o` is created with one property `x`. Creating `o` constructs a new shape that maps the property name `x` to the objects `slot 0`. Creating object `p` with an `x` property uses the same shape mapping as `o`. When property `y` is added to `p` later, the shape associated with `p` is changed to a newly constructed

shape that represents the mapping of `x` to `slot 0` and the mapping of `y` to `slot 1`.

As discussed in Section 2.2, prototype-based object models are useful abstractions for representing types in dynamically-typed languages that can be leveraged to produce efficient code.

2.1.2 Polymorphic Operations

Line 2 in Listing (b) of Figure 2.2 consists of three operations: (1) a property load in `o.score` on the right-hand side; (2) a plus operator in `+`; and (3) a property store in the `=` operator.

Each of these operations is *potentially* polymorphic. (1) and (3) are potentially polymorphic in the number of types of `o` at runtime; how `o.score` is loaded from and stored to depends on the type of `o`. (2) is polymorphic in the number of *combinations* of types of `o.score` and `v`. For example, if `o.score` is an integer and `v` is a string, `+` is a concatenation, if they are both numbers, it is an addition.

For each of these polymorphic operations, the runtime types of `o` and `v` determine which code should be generated for the function. The highlighted operations are *potentially* polymorphic because `updateScore` *could* observe only a single type for `o.score` and a single type for `v` during runtime.

There are three terms commonly used to denote the degree of polymorphism for potentially polymorphic operations. A particular operator, at a particular point throughout program execution, may be: (i) MONOMORPHIC: where only one case has been observed; (ii) POLYMORPHIC: where more than one but less than N cases have been observed; (iii) MEGAMOPRHIC: where N or more cases have been observed. A `case` here describes a set of operand types to an operation. Different cases need to be handled differently by the VM to produce the correct behavior.

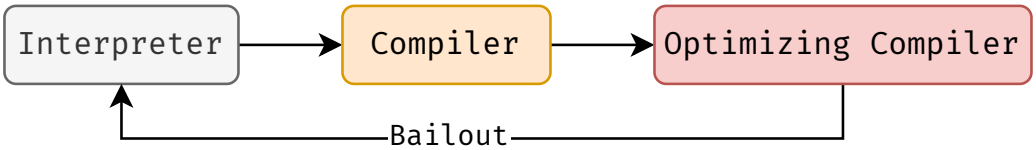

Figure 2.3: Example of langauge VM tiered execution architecture.

2.2 Speculative Compilation

Speculative compilers help address the challenges posed in Section 2.1. At a high level, speculative compilers use profiling to make assumptions about program behavior and infer types dynamically to generate more performant fast-path code. Because the type of inference is based only on assumptions, speculative compilers also need to correctly handle the cases where the assumptions do not hold. Therefore, in speculatively compiled code, the speculated fast path is guarded by a condition based on the inferred types of the variables used in the fast path.

As an example, consider Listing (b) in Figure 2.1. If after several invocations, `updateScore` observes one type for `o` where `o.score` is an integer and `v` is an integer, a language VM can speculate that the types have stabilized – they are unlikely to change – and generate type specialized code for integer addition, guarded by checking the types of `o` and `v`.

2.2.1 Tiered Execution

In dynamically-typed languages, the flow of types through functions needs to be observed at runtime before speculating on the state of type stability. Languages virtual machines typically employ tiered execution models that progressively exploit profiling information to drive speculative Just-In-Time (JIT) compilation decisions. Figure 2.3 shows a common abstract architecture for tiered execution.

The discussion in Section 2.1 highlights the need for dynamically-typed-language virtual machines to be flexible and robust. Initial program execution begins with interpretation. Interpreters execute a single bytecode operation at a time and are flexible and robust to dynamicity. Because the interpreter

does not have the executing bytecode's surrounding context, it appropriately deals with changes in bytecode operand types. Interpreters also have quick start-up times. Compared to compilers, interpreters do not require a distinct compilation step before executing; interpreters can begin executing lines of code immediately. However, due to the interpreter not having context around the executing bytecodes, it is inefficient for an interpreter to frequently execute functions or traces that exhibit type stability.

To counter this, the next execution tier for language VMs is a compilation step. During interpretation, the VM collects profiling information such as the number of invocations of a function and its argument types. With this information, the compiler compiles the function with the appropriate type checks and replaces interpreter execution with a specialized fast path.

Language VMs try to balance the costs of compilation time with the benefit of executing the compiled code. The first compilation step typically trades execution time for compilation speed by foregoing expensive optimizations. Once the VM is *confident enough* that types for a hot function have stabilized, it will compile the function again with an optimization pipeline that mirrors a typical pipeline for statically-typed languages. Because both of the compilers are operating on type-inference assumptions rather than certainty, the compiled function bodies need a *way out* if they observe an unexpected type. In Figure 2.3 this is represented by the arcs labeled `bailout`. A bailout[3] represents the path taken if a condition guarding a compiled fast path fails; the speculation was incorrect, and the executing compiled code must transfer execution back to the interpreter to ensure that the code is handled correctly. When a bailout occurs, it signifies that the code that was speculatively compiled is unable to handle all observed types. Therefore, the compiled function is typically discarded and the interpreter resumes collecting more refined type profiles.

Tiered execution environments ensure that programs have quick start-up times while being able to dynamically produce high-quality specialized code based on type feedback.

[3]Bailouts are also referred to as *deoptimization* or *on-stack-replacement (OSR)*.

2.2.2 Inline Caching

An important technique used to accelerate dynamically-typed program execution in VMs is inline caching. Inline caching was first proposed by L. Peter Deutsch and Allan M. Schiffman in the Smalltalk-80 system to improve the performance of method invocation on dynamically-typed objects [9]. In Smalltalk-80, the expression `o.x` invokes method `x` on a receiver object `o`, and depending on `o`'s runtime type, different methods named `x` could be called [13]. Initially, the call is bound to the default method-lookup routine, which serves as the fallback case. When the lookup routine resolves an address, it overwrites the fallback address, providing a fast path for the observed object's type. On subsequent evaluations of the same expression, the receiver object's type is compared to the cached type corresponding to the fast path, and the fast path is taken if the types match; otherwise, the fallback path is taken, and the call site is re-bound to the newly resolved method address. However, this inline caching scheme has limitations since it only provides a fast path for a single type for each static expression `o.x`.

In contrast, languages such as SELF have more frequent polymorphic method invocations, and Polymorphic Inline Caches (PICs) were introduced by Urz Hölzle, Craig Chambers, and David Ungar to accelerate the operation sites that observe more than one type [7], [15]. Instead of directly embedding the resolved method address into the native code, the system embeds the address of a native code stub that guards on previously observed types and executes the corresponding method. If no guards pass, the fallback path is invoked and a new fast path is appended within the stub.

Modern dynamically-typed language implementations such as SpiderMonkey [30], V8 [33], and JavaScriptCore (JSC) [17] JS engines that are discussed in detail in Chapter 3 use a variation of this PIC technique [34].

In addition to being used to directly accelerate operations, inline caches serve as a very valuable collection of type information by monitoring the types given to an operation. Following the insight from the work by Hölzle et. al. [15], dynamically-typed language VMs take advantage of the type

information provided by PICs to enable the speculative compilation of hot functions that exhibit function-wide type stability.

Chapter 3

Inline Caching in Dynamically-Typed Languages

This chapter discusses in detail the inline caching mechanisms in dynamically-typed language implementations with a focus on JavaScript engines. The end of the chapter provides an alternative view of inline caching in purely interpreted environments, using Python 3.11 [23] as a case study.

3.1 The JavaScript Object Models

As a prototype-based language, JS has an object and inheritance model that follows closely to the description in Subsection 2.1.1. The base prototype of all objects created is the *Object* prototype, a special object whose own prototype is set to `null`, which defines common, built-in behavior for all JS objects[1]. In JS, the *prototype chain* terminates when a prototype slot is `null`. Figure 3.1 shows an example of JS-object prototyping.

JS objects contain properties that can be changed, added, and deleted freely

[1]It is possible to create an object that doesn't inherit any of this behavior with `Object.create(null)`. This is uncommon but worth mentioning.

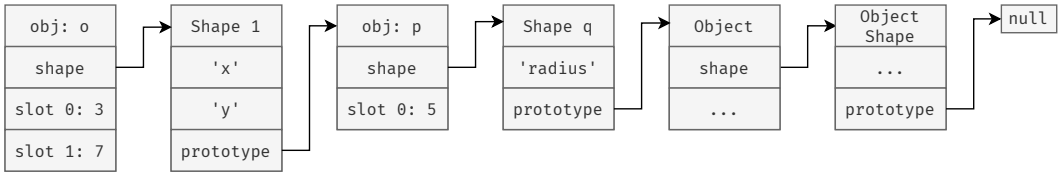

Figure 3.1: JS object model.

at runtime. The in-memory representation of objects differs depending on the implementation; these details will be discussed in the following sections.

3.1.1 Shapes or Hidden Classes or Maps

As mentioned in the Chapter 2, the main feature enabling ICs is the presence of *shapes*. In Figure 3.1, the integrity of the prototype chain of objects is maintained through intermediate shape representations holding the property mappings of slots in the object and the pointer to the prototype. This indirection allows for the same shape to be shared amongst objects with the same mapping and same prototype, reducing memory — as opposed to having a per object mapping — and enabling the IC scheme. Adding properties to an object changes its shape, and the prototype of an object is implied by the shape; objects with the same properties at the same offsets but with different prototypes have different shapes.

The terms *shape*, *hidden class*, and *map* have all historically been used to refer to this idea. In this thesis, shape is consistently used to avoid using the overloaded *map*, and the somewhat misleading and unintuitive *hidden class* terminologies.

3.2 SpiderMonkey

SpiderMonkey is the open-source production JavaScript engine developed mainly by Mozilla [30]. The architecture of SpiderMonkey has and will continue to evolve; the description below outlines the state of SpiderMonkey in the Firefox 111 release.

3.2.1 Execution Tiers

In SpiderMonkey, JavaScript source code is compiled Just-In-Time (JIT) into a collection of *scripts* that correspond to executable bodies such as functions, closures, modules, or `eval` statements. Initially, the scripts are compiled to bytecode, but may gain additional representations as they transition between different execution engines:

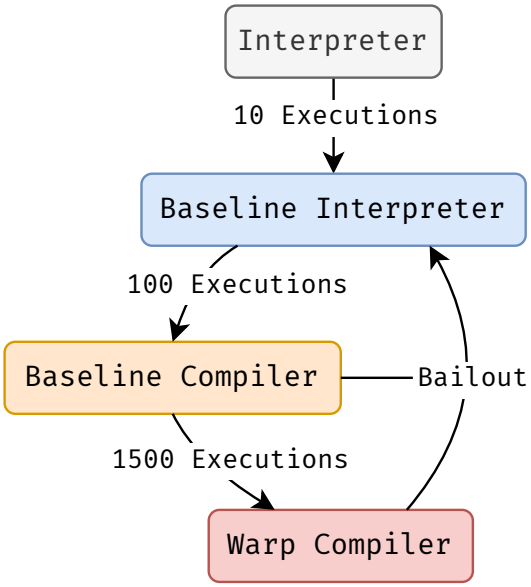

Figure 3.2: SM execution tiers and transition conditions.

Interpreter: For a script's initial invocations, SpiderMonkey employs a single-bytecode interpreter.

Baseline Interpreter: The next tier, the Baseline Interpreter, starts collecting bytecode operand-type information by utilizing ICs for supported bytecode operations.

Baseline: Then, a script may be compiled by the Baseline compiler, a JIT compiler that stitches together native code sequences for bytecodes. For IC-supported operations, Baseline emits a call to the compiled IC code.

Warp and Ion: For performance-critical scripts, SpiderMonkey employs Ion, a type-specializing and speculative compiler that generates executable code optimized to the type information stored within ICs. In the case of a speculation failure, Ion-compiled code will *bailout* to the Baseline Interpreter tier to collect more refined profiling data.

3.2.2 SpiderMonkey's Inline Caches

In SpiderMonkey, ICs operate at a bytecode level such that each supported bytecode site gets its own IC. All ICs are PICs; each IC is a linked list of

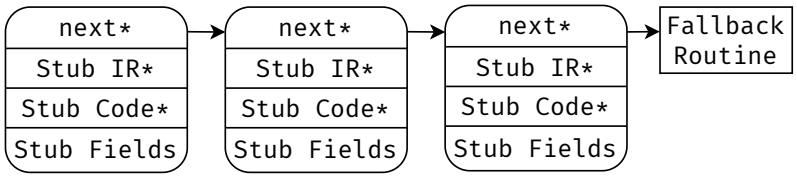

Figure 3.3: Inline Cache structures in SpiderMonkey.

stub structures. There are two types of stub data structures: *specialized stubs* and *fallback stubs*. They both contain pointers to associated machine code – the *stub code* – and a counter variable keeping track of how many times the stub has been entered. Specialized stubs hold IC information for each distinct case observed at an IC-supported bytecode site including associated metadata such as a pointer to the stub's IR and the data on which the stub operates – the *stub fields*. Fallback stubs contain the ICs *state* and their stub code calls the associated fallback handler with the necessary logic to produce a result for the bytecode and the ability to attach new stubs to the stub chain. Each IC-supported bytecode site is initialized with a fallback stub corresponding to the bytecode operation. Figure 3.3 shows a high-level overview of these structures.

Inline-Cache States

There are four[2] states in SM in which ICs can exist, each coarsely characterizing the observed type history for a given IC:

UNLINKED: if no stubs have been attached other than the fallback stub, suggesting the code path has yet to be visited.

SPECIALIZED: if 1 to N specialized stubs have been attached, suggesting a monomorphic or low-degree polymorphic site.

MEGAMORPHIC: if more than N specialized stub attachments would be needed to handle all the observed cases, suggesting a high-degree polymorphic site.

[2]Three in the source code, but the unlinked state is handled differently in certain parts of the engine so it is included separately here.

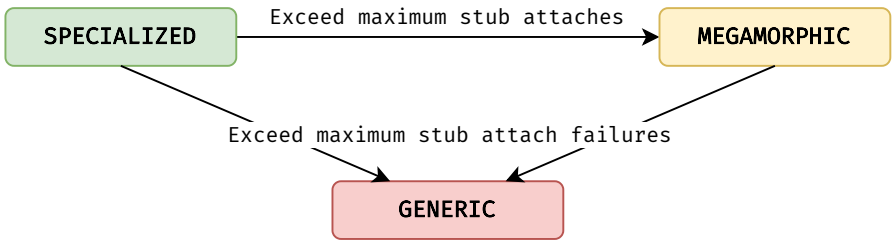

Figure 3.4: SpiderMonkey's IC states and transition criteria.

GENERIC: if there are multiple failures attaching stubs, suggesting unsupported IC cases.

Stubs attached in SPECIALIZED mode are the most performant stubs that are type specialized for specific cases. Stubs attached in MEGAMORPHIC mode each handle a variety of cases by relaxing type specialization but often require a slower, more general technique to compute the correct result of the operation. Stubs attached in GENERIC mode revert to a VM call similar to the direct interpretation of the bytecode operation. Chapter 4 provides a more detailed view of IC generation policies in each of these states.

Each IC chain is initially UNLINKED, from which point it may either (i) transition to SPECIALIZED, (ii) transition to MEGAMORPHIC, or (iii) transition directly to GENERIC. When the IC chain transitions from one mode to another, it discards all the currently attached stubs other than the fallback stub. As shown in Figure Figure 3.4, an IC chain transitions from SPECIALIZED to MEGAMORPHIC when it exceeds the maximum number of attached stubs that are allowed for a specialized IC chain. The goal of this limit is to prevent the creation of long chains of stubs whose traversal may cause high overhead. In SpiderMonkey, this limit is set to 6 stubs. Each IC chain also keeps track of the number of times a fallback stub has been reached and failed to attach a stub. Failure to attach any stub can happen because JavaScript is an extremely dynamic language with a large number of possible behaviors, many of which are effectively never seen in practice. As a result, optimization effort is typically focused on observed patterns, leading to rare combinations of operations having no IC developed for them. In addition, there is a maximum number

of failures-to-attach that the stub chain will tolerate; when that is hit, the stub chain transitions to GENERIC mode to avoid wasting further resources on attempting to attach new stubs. Figure 3.4 displays the IC states and directional transition conditions between states.

Megamorphic Property Access Operations

For property lookup operations, MEGAMORPHIC stubs rely on a typical software cache – the *Megamoprhic Cache* – that maps the hash of a `(shape, property key)` pair to the property in the object. The Megamorphic Cache is a global cache that has 1024 entries. It is a single-level cache that handles collisions by overwriting previous entries. MEGAMORPHIC stubs call a VM function that performs a cache lookup on the Megamorphic Cache. If there is a cache hit such that the shape of the incoming object is the same as the cached object, it returns the correct offset into the object to access a given property.

CacheIR

In 2016, the SM team rehauled the IC infrastructure by developing a novel IR for ICs called CacheIR [20]. CacheIR is a simple bytecode specialized for compiling IC stubs. CacheIR gives a useful abstraction that enables inline cache code sharing and speculative compilation by lowering the IR in optimizing compiler tiers. Chapter 4 provides a detailed description of CacheIR and the virtual machine architecture it enables.

3.3 JavaScriptCore

JavaScriptCore (JSC) is the open-source production JavaScript engine developed mainly by Apple for the WebKit browser engine [17]. The description below outlines the state of JSC in the WebKit 173 release.

3.3.1 Execution Tiers

JSC uses a register-based bytecode as the source of truth throughout all execution tiers.

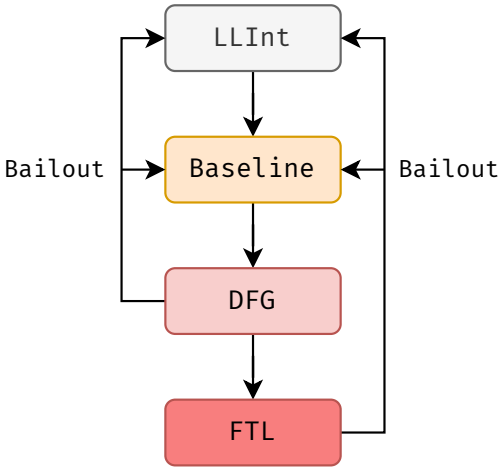

Figure 3.5: JSC execution tiers.

LLInt: LLInt — or Low-Level Interpreter — is the initial execution tier. It executes the program by interpreting one bytecode at a time and collects bytecode operand type information that is used in the higher tiers.

Baseline: Once LLInt has collected enough type information for a function, it is compiled using the Baseline JIT. Baseline is a template JIT compiler that compiles entire functions at a time. Each bytecode is compiled into a predefined sequence of instructions.

DFG: DFG (for data flow graph) is a JIT tier that performs speculations based on the profiling data collected through LLInt and Baseline. DFG converts bytecodes into an internal IR called *DFG IR* that allows for inter-bytecode reasoning to enable compiler optimizations. A primary goal of DFG is maintaining low compilation times, at the cost of generating sub-optimal code. Because of this, DFG does not perform more robust but time-consuming optimizations such as global register allocation, escape analysis, loop optimizations, and any optimization that relies on an Static Single Assignment (SSA) IR. When a speculation fails, DFG can bail out to either LLInt or Baseline to resume execution with fewer assumptions.

FTL: FTL — or Faster Than Light — is the top-tier optimizing compiler, maximizing throughput at the cost of increased compilation times. FTL builds on the pipeline and infrastructure of DFG by reusing many of DFG's

optimizations while also opting into the aforementioned time-consuming optimization that DFG does not perform. In addition to DFG IR, FTL also uses DFG SSA IR, B3 IR, and Assembly IR. When a speculation fails, FTL can also bail out to either LLInt or Baseline.

3.3.2 JavaScriptCore's Inline Caches

LLInt uses inline caches for property accesses by caching a shape and property offset as metadata on the property access operation. The cached shape and property offset are updated each time the operation encounters an object with a new shape. Unlike SpiderMonkey, which relies on creating IC chains consisting of distinct stubs to handle polymorphism, JSC's Baseline ICs rely on a technique called repatching. Repatching is also the technique used in the classic inline caching implementations from the work on SELF [7], [15] and Smalltalk-80 [9], as discussed in Subsection 2.2.2. With repatching, Baseline allocates slabs of memory to hold machine code for each IC. Each IC slab is initially an unconditional `jump` instruction to the VM routine—the slow path—followed by a sequence of `no-op` instructions. When Baseline observes type information for an IC-supported operation, it generates type-specialized code guarded by the observed types and a fallback path with the `jump` to the slow path. That code is stored in the slab for the IC, overwriting the initial `jump` to the slow path and `no-op` instructions. Whenever a new set of types is observed, the existing code is updated to handle the new case.

JavaScriptCore's inline caches use data structures to: (i) store the object shapes observed by each property access operation, and (ii) indicate whether the IC is polymorphic or megamorphic.

DFG's ICs also employ repatching, but DFG ICs rely on profiling information collected in ICs from LLInt and Baseline. DFG checks if the LLInt IC and the Baseline IC have the same monomorphic case. If they do, DFG represents the IC as an inline sequence of DFG IR operations to enable context-aware reasoning about ICs, such as guard elisions between inlined ICs. If the degree of polymorphism reported by Baseline is below a set threshold, DFG also represents the IC as a sequence of inline DFG IR operations that handle low

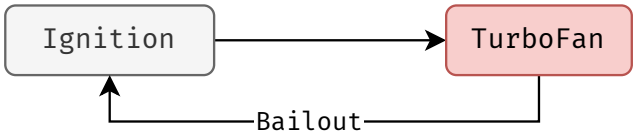

Figure 3.6: V8 execution tiers.

degrees of polymorphism. Otherwise, if the degree of polymorphism is below the maximum-polymorphic-case threshold, DFG reinitializes a fresh IC slab to continue collecting refined profiling data.

FTL generates internal IR and machine code based on IC analysis from LLInt, Baseline, and DFG ICs. If an IC is already inlined in DFG IR (i.e., monomorphic) or low-degree polymorphic represented by a DFG IC, FTL treats the DFG IR operations corresponding to the IC the same as any other sequence of DFG IR, lowering the operations to equivalent B3 IR nodes.

Megamorphic Property Access Operations

For operations that exceed the maximum-polymorphic-case threshold, Baseline, DFG, and FTL create an IC that uses a Megmamorphic Cache similar to SpiderMonkey. The execution engine repatches the inline cache to call a VM function that performs a cache lookup on the Megamorphic Cache. Unlike SpiderMonkey, JSC's Megamorphic Cache is multi-level and has a more sophisticated cache collision mechanism. There are two cache levels, primary and secondary. When a property access maps to a primary cache entry, JSC moves the property access currently mapped in the primary cache to the secondary cache.

3.4 V8

V8 is the open-source production JavaScript engine developed mainly by Google for the Chromium browser engine [33]. The description below outlines the state of V8 in the 10.0 release.

3.4.1 Execution Tiers

Ignition: Ignition is an interpreter that is responsible for lowering JS source to bytecode and interpreting a single bytecode at a time. Ignition profiles the program during execution and stores type information for operations like property accesses and binary arithmetic and uses inline caches to accelerate interpretation.

TurboFan: For hot functions, V8's optimizing compiler, TurboFan, consumes the bytecode and type information for the inline caches and speculatively generates type-specialized and highly-optimized code.

V8's tiering structure, as shown Figure 3.6, is significantly simpler than JSC and SM. Despite this simplicity, V8 is a very high-performance JavaScript engine and is used in the popular Node.js [10] backend. The differences in tiering structure between engines highlight the flexibility and freedom in VM design for languages that use speculative compilation.

3.4.2 V8's Inline Caches

The main component of V8's inline caches is a data structure called a *Feedback Vector*. For each function, Ignition creates an entry in a corresponding Feedback Vector for each IC-supported bytecode operation. For example, a Feedback Vector entry can correspond to an IC for a property access, call, or a binary arithmetic operation. The Feedback Vector entry keeps track of the state of the IC, which is either: (i) Uninitialized; (ii) Monomorphic; (iii) Polymorphic; or (iv) Megamorphic. Additionally, each Feedback Vector entry also contains data corresponding to an observed set of operands. For example, an entry for a monomorphic property access IC includes the shape of the object and the offset of the property on the object. Ignition uses a precompiled set of routines that correspond to each IC-supported operation. Each of these routines operates on the Feedback Vector entry to compute the result of the IC-supported operation. When compiling a function, TurboFan exploits the information in the functions Feedback Vector to generate highly-specialized machine code.

Megamorphic Property Access Operations

Similar to JSC, megamorphic property access operations in V8 use a multi-level Megamorphic Cache. However, V8's cache maps property access keys to handlers used to compute the access rather than to property offsets. When a Feedback Vector entry indicates that a property-access IC is megamorphic, Ignition envokes a Megamorphic Cache lookup. On a cache miss, the new property access is inserted into the cache. If there is a collision, the primary cache's existing entry is moved to the secondary cache, overwriting the secondary cache entry. Cache hits return the handle to compute the appropriate property access.

3.5 Interpreter-Only Inline Caching with Python 3.11

With the release of the CPython implementation of Python 3.11 in late 2022 [23], an IC scheme known as *Quickening* became widely available to accelerate interpretation. Unlike SELF and JavaScript implementations, CPython is purely interpreted and ships without any kind of dynamic translation engine or JIT compiler. Therefore, for inline caching to be useful, it must not rely on compilation of native code at run-time.

Quickening is a technique used in language VMs to specialize VM function dispatch with respect to the operand types of a bytecode. Performing inline caching with Quickening was initially presented in the work by Brunthaler [5], providing an implementation in Python as a proof-of-concept. Take the example of `a + b`, in Python, like in JavaScript, the types of `a` and `b` may change during execution. Rather than compiling a specialized machine code stub akin to Deutsch and Schiffman [9], the default VM call to perform a slower `+` operation on generic inputs is replaced by a specialized VM function that expects the operands to be of the same types as `a` and `b`. After specialization, the VM will dispatch to the *quickened* version of the function as long as the types stay consistent. If the observed types change, an alternative quickened function replaces the previous one; therefore, Quickening is a type of monomorphic

inline cache.

Quickening was introduced as part of what the CPython group is calling a *Specializing Adaptive Interpreter* in PEP 659 [26]. Operations that are candidates for Quickening include calls, attribute loads from objects, attribute loads from the global namespace, and binary arithmetic. Each Quickening candidate operation has a *family* of functions that are shipped with the interpreter. Each member in the family contains a function specialized for certain input operand types. Quickening has substantial performance implications, consistently providing speedups of 10% to 60% over disabling the optimization. Moreover, the architecture of Quickening for purely interpreted dynamically-typed language implementations is validated by its inclusion in CPython.

Chapter 4

The Benefits of a Structured Representation for Inline Caches

Most software stacks that support inline caching use low-level, often *ad-hoc*, IC data structures for code generation. This chapter discusses *CacheIR*, a design for inline caching built entirely around an intermediate representation (IR) which: (i) simplifies the development of ICs by raising the abstraction level; and (ii) enables reusing compiled native code through IR matching techniques. Moreover, this chapter describes *WarpBuilder*, a novel design for a Just-In-Time (JIT) compiler front-end that directly generates type-specialized code by lowering the CacheIR contained in ICs; and *Trial Inlining*, an extension to the inline-caching system that allows for context-sensitive inlining of context-sensitive ICs. The combination of CacheIR and WarpBuilder have been powerful performance tools for the SpiderMonkey team, and have been key in providing improved performance with less security risk.

CacheIR was designed and implemented by the SpiderMonkey development team at Mozilla and open-source contributors to the SpiderMonkey project. The purpose of this chapter is to provide a detailed view into a modern inline caching design and evaluate WarpBuilder by comparing it to SpiderMonkey's previous design.

```
1 GuardToObject InputId0 --> ObjId0
2 GuardShape ObjId0, Field0
3 LoadFixedSlotResult, ObjId0, 8
4 ReturnFromIC
5 --Stub Fields---
6 Field0: Shape 0xabcdef0123
```

Figure 4.1: An example of CacheIR for an object property read: `obj.prop`

4.1 A Linear Bytecode for Inline Caches

CacheIR is a simply typed bytecode specialized for compiling inline caches. CacheIR bytecode is 'linear' in that it has only two control-flow primitives:

- **Guards**: Instructions that verify a stub invariant, preventing the execution of the stub if the guard does not hold.
- **Return**: A single bytecode that returns from the IC stub code.

Containing no other control flow instructions, the intermediate representation of an inline cache in the CacheIR is akin to an extended basic block [21]. Each IC has a single entry point, has multiple exit points through guard failure paths and the return operation. Once the execution passes all the guards, and all instructions in the IC execute once in order.

CacheIR bytecodes operate on typed *Operands*, which are either input values or the return value of a CacheIR bytecode operation. The number of implicit input operands in the CacheIR for an IC is determined by the arity of the bytecode to which the IC is attached. For an IC attached to a bytecode that produces a value, the CacheIR has an output operand for that value as well. In addition to operands, CacheIR stubs have *stub fields*, which are values associated with and used within the stub. For Baseline ICs, stub fields facilitate the sharing of native code for stubs that are identical except for offsets and pointer values, and simplify the process of integrating stubs into the garbage collector.

Figure 4.1 shows an example of a CacheIR illustrating guards, operations, and stub fields. The `GuardToObject` op is an example of a guarded cast; either the input is of Object type, producing a new Object typed operand, or the guard condition fails and control is given to the next stub. The `GuardShape` op tests if the object-operand's shape matches the shape stored in the stub field. `LoadFixedSlotResult` loads a value out of the fixed slot at offset 8 in the object[1], placing the result into the implicit result register (hence the `Result` suffix).

Currently, SpiderMonkey's CacheIR has more than 300 CacheIR instructions, including 64 guard conditions, covering a large number of behaviors inside the engine. Moreover, the IR's simplicity enables a straightforward implementation path to add support for further operations.

4.1.1 Generation of CacheIR

CacheIR is generated in the fallback path for an IC miss. While details vary gently, the fallback handler logic generally operates as follows: (i) compute the result for the current operation; (ii) invoke the appropriate CacheIR IR Generator — this generator analyses the input values, opcode, and resulting value, and uses a simple hand-crafted pattern matching code to instantiate a matching sequence of CacheIR operations that handle the input values; (iii) generate native code from the sequence of CacheIR operations; (iv) attach the resulting stub to the front of the IC chain.

Raised Abstraction == Higher Productivity

Language runtime development must focus on performance improvement. Sometimes a performance cliff can be eliminated by designing the engine to generate specialized code for specific cases. SpiderMonkey, for example, often generates specialized code by adding support for a specific case in the inline cache generator. For instance, consider an illustrative example: Assume that a performance analysis determines that an important site often performs arithmetic with `null`.

[1]This slot was determined during the generation of this cache and is correct because of the shape guard previous.

Such operations are unexpected and thus are not handled by the default IC generation policy. CacheIR simplifies the process of inserting an IC to generate specialized code for the execution of operations such as `a + b` where either `a` or `b` is `null`, leading to performance improvement in this example.

Adding such support often only requires a modification to the existing Binary Arithmetic CacheIR generator[2] that adds a case to emit the correct CacheIR, as shown in Figure 4.2. The general-shape CacheIR generation pattern matches on the actual results that the fallback stub observed when running the operation (lines 2-9), and then generates CacheIR that provides a fast path for the specific observed case (lines 17-21). In some cases, correct handling may require adding a CacheIR operation. In SpiderMonkey, adding a CacheIR operation requires a modification of an operator description file, and code generation for that operation through a platform-independent MacroAssembler.

By adding support to the CacheIR generator, we've also added support for specializing this operation to Warp, assuming the CacheIR operations are all successfully handled by Warp.

Working with CacheIR raises productivity for writing ICs in other ways as well. CacheIR has a simple register allocator to allow managing values inside of caches. Furthermore, CacheIR also automatically creates the failure paths required to restore the input registers to their original values on failure allowing every stub in the IC chain to start from the same state.

Using an Intermediate Representation for inline caches also eases the development of tooling to analyze the behaviour of inline caches, since a structured machine-independent representation is appreciably easier to investigate than generated native machine code.

Stub Generation Policies: Avoiding Pathological Outcomes

Each IC chain has some associated state: the number of stubs attached, and a *mode*. There are three modes, coarsely characterizing observed type history for an IC chain:

[2]Used for infix binary operations.

```
1  AttachDecision BinaryArithIRGenerator::tryAttachNullInt() {
2    // Only Handle Add
3    if (op_ != JSOp::Add) {
4      return AttachDecision::NoAction;
5    }
6    // Only handle LHS null RHS int32.
7    if (!lhs_.isNull() || !rhs_.isInt32()) {
8      return AttachDecision::NoAction;
9    }
10
11   ValOperandId lhsId(writer.setInputOperandId(0));
12   ValOperandId rhsId(writer.setInputOperandId(1));
13
14   // null + int32rhs = int32rhs
15   writer.guardIsNull(lhsId);
16   Int32OperandId rhsIntId = writer.guardToInt32(rhsId);
17   writer.Int32Result(rhsIntId);
18   writer.returnFromIC();
19 }
```

Figure 4.2: A simplified fictional example of the CacheIR generation process for a `null` + int opportunity

1. Specialized: In this mode, the CacheIR generators attempt to create stubs that are tightly specialized to the observed values, but can only handle specific, tightly guarded cases. For example, a specialized stub for a property read has a guard on the receiver being an object and a guard on the object's shape (as in Figure 4.1).

2. Megamorphic: In this mode, the CacheIR generators create stubs that are not as type specialized as Specialized stubs. Often this involves creating a stub that calls a routine in the VM runtime, which is slower than a Specialized stub, but still faster than interpretation. For a property read, Megamorphic stubs guard on the receiver being an object before calling a VM routine to produce the result via a lookup in a global property cache.

3. Generic: In this mode, stub attachment is disallowed, and a call to a VM runtime routine will simply provide the result value without attempting

to attach a stub.

Each IC chain is initially SPECIALIZED, from which point it may either (i) remain SPECIALIZED, (ii) transition to MEGAMORPHIC, or (iii) transition directly to GENERIC. When the IC chain transitions from one mode to another, it discards all the currently attached stubs other than the fallback stub. As shown in Figure 3.4, an IC chain transitions from SPECIALIZED to MEGAMORPHIC when it exceeds the maximum number of attached stubs that are allowed for a specialized IC chain. The goal of this limit is to prevent the creation of long chains of stubs that may cause high overhead to traverse. Each IC chain also keeps track of the number of times a fallback stub has been reached and failed to attach a stub. Failure to attach any stub can happen because JavaScript is an extremely dynamic language with a large number of possible behaviors, many of which are effectively never seen in practice. As a result optimization effort is typically focused on observed patterns, which can lead to rare combinations of operations having no IC developed for them. In addition, there is a maximum number of failures-to-attach that the stub chain will tolerate; when that is hit, the stub chain transitions to GENERIC mode to avoid wasting further resources on attempting to attach new stubs.

4.1.2 Compilation of CacheIR into Native code

SpiderMonkey has two CacheIR compilers: one compiler shared by Baseline Interpreter and Baseline, called the *Baseline CacheIR Compiler*, and one specialized to Ion — SpiderMonkey's top-tier compiler used only for the hottest scripts — called the *Ion CacheIR Compiler*. The two compilers share a considerable amount of engine code but are specialized to the required calling convention of the target IC system. Each compiler also adopts a different policy for the handling of Stub Fields.

The Baseline CacheIR compiler handles stub fields by generating native code that loads the values out of the stub metadata. The Ion CacheIR compiler adopts a policy of directly embedding stub field values in the IC stub code. Field-value embedding makes Ion IC stub code ineligible for sharing but requires

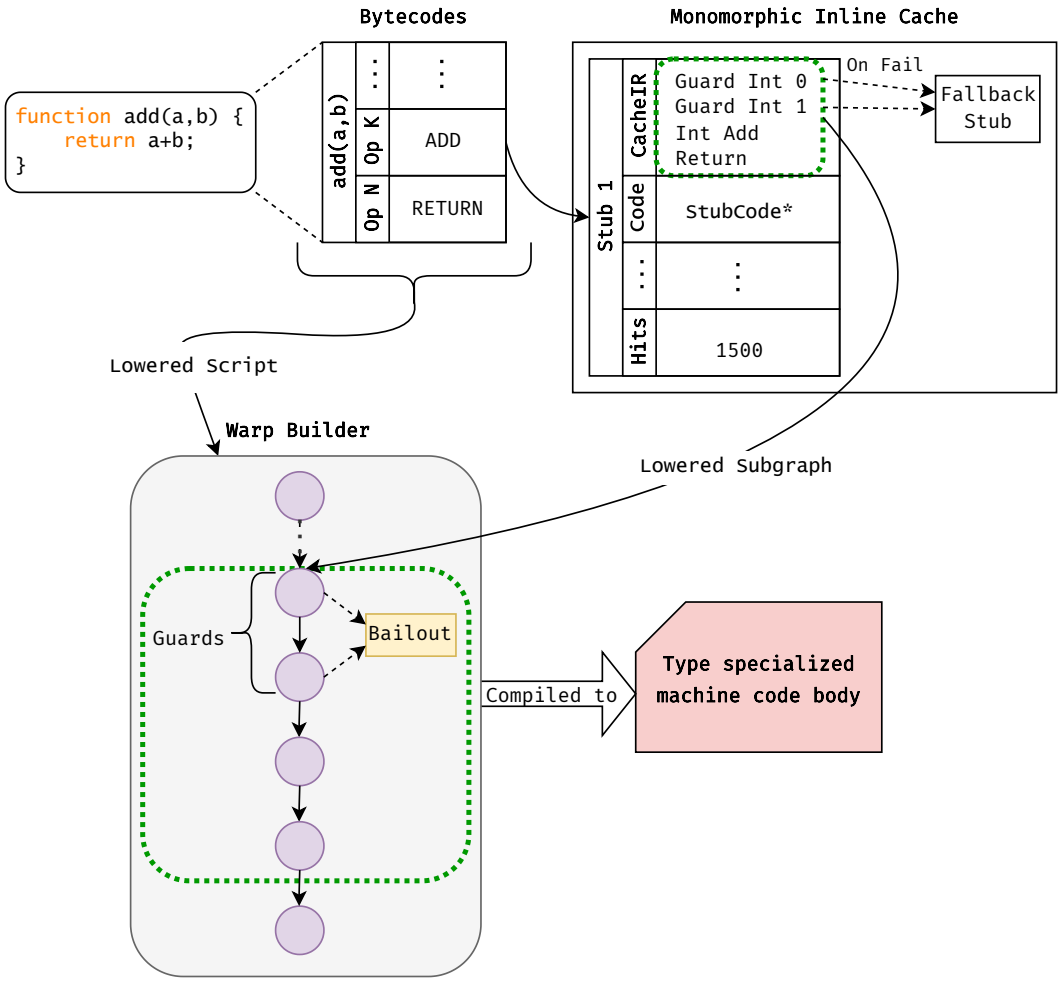

Figure 4.3: A diagram of the CacheIR powered transformations in SpiderMonkey

less indirection, and thus Ion ICs ultimately execute faster than Baseline ICs. The Baseline CacheIR compiler's approach is slightly less performant than Ion's approach but has the benefit that caches with the same CacheIR that differ only in the values of their stub fields can share native code. Stub code sharing dramatically reduces the amount of native code required by Baseline ICs, saving memory and the time required to generate native code when stubs can be shared. Furthermore, stub code sharing also avoids a major cost of dynamic code generation which is the page table manipulation required to mark pages as executable at runtime.

Both CacheIR compilers use a simple register allocator to track where each operand is during execution — in a register, in a stack slot, on the language stack, etc. This register allocator can provide extra registers to caches where required. If a guard fails, the IC needs to restore the input registers to their original values, allowing the next stub in the chain to start in a known state. The register allocator has the information that it needs to generate these failure paths automatically. Failure paths are shared between guard instructions if the register state has not changed between the guards.

4.1.3 Dynamic Specialization Without Dynamic Allocation of Executable Pages

JIT Compilers require the ability to write to pages marked executable. This is sometimes prohibited by platforms (often to improve security), and as such SpiderMonkey, if run on one of these platforms, must necessarily limit itself to the interpreter. CacheIR provides a potential road forward to maintain a specializing IC system that does not rely on writing new code. Since SpiderMonkey's IC scheme already supports sharing native code for stubs which have the same CacheIR, pre-compiling native code for the N most common stubs and shipping them with the engine binary would provide the speedups of inline caching, without the requirement for writing executable code at runtime[3].

[3]In SpiderMonkey this would *also* require pre-compiling the Baseline Interpreter, as it is only the Baseline Interpreter and higher tiers that can attach IC stubs

4.2 WarpBuilder: Consuming CacheIR Directly as Type Feedback

WarpBuilder, a front-end for the Ion Compiler, converts bytecodes into MIR and was built around the insight that CacheIR provides an excellent source of type information for an optimizing compiler.

It replaces a component called IonBuilder, which used a hybrid Type Inference (TI) system, proposed by Hackett et al., that uses both static-analysis and runtime-collected type information to infer facts about object types and shapes [14]. The TI system allows for complicated reasoning about objects and nested property accesses. However, its power came at a cost: Type Inference consumed memory to power optimizations which could only occur once functions made it to top-tier compilation, but an even larger cost was the engineering costs associated with TI.

To be sound, the TI system needed information to be correctly propagated throughout the engine, which meant that TI code was necessary for many parts of the engine. Moreover, any failure to properly maintain type data could lead to security problems because the Ion optimizing compiler would consult the TI system and use the provided invariants to elide checks that would otherwise be necessary. This was particularly pernicious because erroneous handling of a value in one place could be exploitable by code very far away, as a result of the poisoning of the global analysis.

In WarpBuilder, type data is exclusively sourced from the inline caches generated as part of lower-tier executions, and the consumption of that type data is local to the MIR for a particular bytecode. WarpBuilder precisely builds specialized code by directly compiling the CacheIR for an eligible stub to MIR, converting guard failures to bailouts. With this compilation strategy, the MIR code is immediately specialized to the observed types in the program, and those type checks are directly made visible to the Ion optimizer, which is able to re-order them, and eliminate redundancy to further optimize the code. Furthermore, more complex ICs with complicated guard conditions do not then require more complicated analysis glue code in WarpBuilder to take

advantage of stored types – so long as Warp is able to compiler the CacheIR instructions, the guards are built correctly and automatically every time.

Compared to IonBuilder it is an appreciably simpler approach, eliminating the requirement for globally correct reasoning, and limiting the scope of impact for erroneous code. Moreover, using CacheIR for ICs in lower tiers is a performance optimization that provides direct value, whereas tracking TI information is pure overhead before tiering up to the Ion compiler. Therefore, WarpBuilder also provides improvement over IonBuilder on memory consumption, startup time, and flat code profiles

4.2.1 The Compilation Pipeline

The WarpBuilder compilation process is split into three phases:

Phase 1: Snapshot Building

At the start of top-tier compilation a *WarpOracle* creates a snapshot of information, including the CacheIR stub information for the script. The WarpBuilder uses this snapshot to generate MIR.

Adopting this snapshot-generation process allows the remainder of the Warp compilation to be executed in a different thread from the main thread. This latency optimization is important because the main thread performance is extremely performance-sensitive in a production JS engine.

Phase 2: MIR Generation

In a separate thread, the WarpBuilder uses the bytecode and the WarpSnapshot, to create MIR. During this phase, when a bytecode that supports ICs is encountered, Warp can do one of the following:

1. If the IC has multiple active stubs, Warp emits code that constructs and uses an Ion IC chain.

2. If the IC has only a single stub attached, or has only a single active stub at the front of the IC chain, then Warp lowers[4] that stub's CacheIR to

[4]This step is referred to as *transpilation* in the source code.

MIR and inline.

3. If no IC stubs are attached, Warp generates an unconditional bailout, thus enabling speculative dead code elimination while maintaining appropriate handling if that speculation is false.

Stub lowering is the relatively straightforward process of converting the sequence of CacheIR ops in the stub into equivalent MIR nodes. Rather than jumping to another implementation or stub, **Guard** instructions are lowered such that guard failure results in a bailout, and execution will continue in the baseline interpreter.

Previous work used inline caches to drive optimization by taking advantage of the insight that inline caches capture exactly the set of runtime observed types at a particular bytecode [16]. However, unlike previous work that simply refined types using the data contained in ICs, SpiderMonkey lowers CacheIR and generates specialized code in Warp without the engineering cost of writing, yet again, type-specialization code for the Ion Compiler, thus allowing for direct fast-path code generation. Because lowering CacheIR to MIR is a mechanical translation, Warp also does not need to understand source language level semantics; single CacheIR operations are lowered individually to a predictable MIR node. A lowering pipeline for a CacheIR operation is also reused between different kinds of ICs. For example, for a GuardShape CacheIR operation, the pipeline needs to be defined once then it can be used in any operation backed by an IC that needs to guard an object's shape.

Phase 3: Optimization and Code Generation

The MIR is then optimized by the Ion optimizer, lowered into LIR, then compiled into native code.

4.2.2 Trial Inlining

True monomorphism is highly desirable because it allows for tightly specialized code generation and it provides a reasonable basis for making type-based assumptions in the compilation pipeline. ICs must reflect, as accurately as

```
1  function adder(a,b) {
2      return a+b;
3  }
4
5  function strings() {
6      var str = "";
7      for (var o of ["a","b","c"]) {
8          str += adder(o,",");
9      }
10     return str;
11 }
12
13 function numbers() {
14     var n = 0;
15     for (var o of [1,2,3]) {
16         n += adder(2,o);
17     }
18     return n;
19 }
```

Figure 4.4: An example of local monomorphism: the ADD op within `adder` may operate either on strings on integers; however, within each call context the types that reach the ADD are monomorphic.

possible, the expected set of types during Warp compilation because type specialization is handled by IC analysis. An obvious challenge is how to handle the inlining of functions that may be polymorphic across the program, but monomorphic in a particular call context (see Figure 4.4).

To handle this case, the SpiderMonkey team developed *Trial Inlining*. Trial Inlining allows the Baseline compiled code to associate distinct sets of ICs to distinct call sites that call the same function. After trial inlining, each set of ICs collects type information local to the call site that the set is associated with. Therefore, if Warp inlines that call, it can create type-specialized code for that call site, handily exploiting local monomorphism. Trial inlining can nest within another trial inlining, further specializing code of calls within the Trial Inlined calls[5].

[5]To avoid unbounded memory consumption, however, nesting is limited to a maximum inlining depth of 4.

4.3 CacheIR and WarpBuilder Evaluation

An evaluation of a point in the design space of dynamic language compilation is necessarily partially empirical and partially subjective. In this section, CacheIR and Warp are evaluated on a number of dimensions to support that: (i) CacheIR is a useful abstraction for developing inline caching systems; and (ii) CacheIR creates the conditions for a JIT compiler that has a simple, high-level design that nevertheless provides excellent performance. .

4.3.1 Benchmarks, Hardware, and Software

Data presented in this section was collected from a machine that runs Fedora 36 (Kernel 5.18.11-200) and is equipped with an Intel i5 12400 16 GiB of DDR4 memory. Moreover, this section uses Speedometer 2.1 – SPEEDOMETER – and JetStream 2.1 – JETSTREAM – benchmark suites to evaluate performance, SPEEDOMETER as a proxy for real-world workloads, and the AreWeSlimYet (AWSY) [2] *tp6* benchmark to evaluate memory consumption.

SPEEDOMETER contains 16 subtests, all of which are `To-Do list` applications written in different popular JavaScript frameworks like React, React with Redux, Ember.js, Backbone.js, AngularJS, Vue.js, jQuery, Preact, Inferno, and Flight. These frameworks are ubiquitously used in Web development and SPEEDOMETER is meant to reflect real-world Web-App workloads by conducting operations on a `To-Do list` both synchronously and asynchronously. To characterize performance, SPEEDOMETER computes a final score:

$$\frac{6000}{geomean(medians) * correctionFactor} \tag{4.1}$$

where *medians* is the set containing a value for each subtest, computed by the median run time in milliseconds across all iterations; and *correctionFactor* is a scalar value used to scale down the final score. Faster subtest run times — smaller geometric mean values — increase the final score.

JETSTREAM contains 64 subtests evaluating computationally intensive workloads.[6] On each invocation of JetStream, each subtest runs for 120

[6]A full list is found here: https://browserbench.org/JetStream2.1/in-depth.html

iterations and computes a score value:

$$subScore_i = \frac{5000}{time_i} \tag{4.2}$$

where $time_i$ is the time in milliseconds to complete the i^{th} iteration. JetStream computes a single score across all iterations for each subtest B:

$$score_B = geomean(firstIteration_B, worstFour_B, average_B) \tag{4.3}$$

where $firstIteration_B$ is $subScore_0$; $worstFour_B$ is the arithmetic mean of the lowest four *subScore* values; and $average_B$ is the arithmetic mean across all iterations. To characterize overall performance, JetStream computes a final score:

$$geomean(scores) \tag{4.4}$$

where *scores* is the set containing the $score_B$ values for each subtest.

AreWeSlimYet is a project maintained by Mozilla to track memory usage across Firefox builds. The tp6 test automates opening browser tabs and loading popular Web pages to simulate common user behavior. AWSY collects memory usage statistics that discriminate between sources of memory consumption. This evaluation focuses on the memory usage attributed to the SpiderMonkey JavaScript engine. Each memory usage value is the geometric mean across all iterations – in this evaluation 15 – of an AWSY run.

4.3.2 Experimental Methodology

The results presented in Sections 4.3.3 and 4.3.4 use Mozilla's `mach` tool with the `raptor` subcommand to collect benchmark metrics for Speedometer, JetStream, and the `awsy-test` subcommand to collect metrics for AWSY. Each data point in the figures represents the metrics described above computed from 15 `mach` invocations for both Speedometer and JetStream.

4.3.3 Execution Engines, Inline Caches and their Impact on Performance

Figure 4.5 examines the performance contribution of various execution engines on Speedometer and JetStream. The speedup factor of each tier is

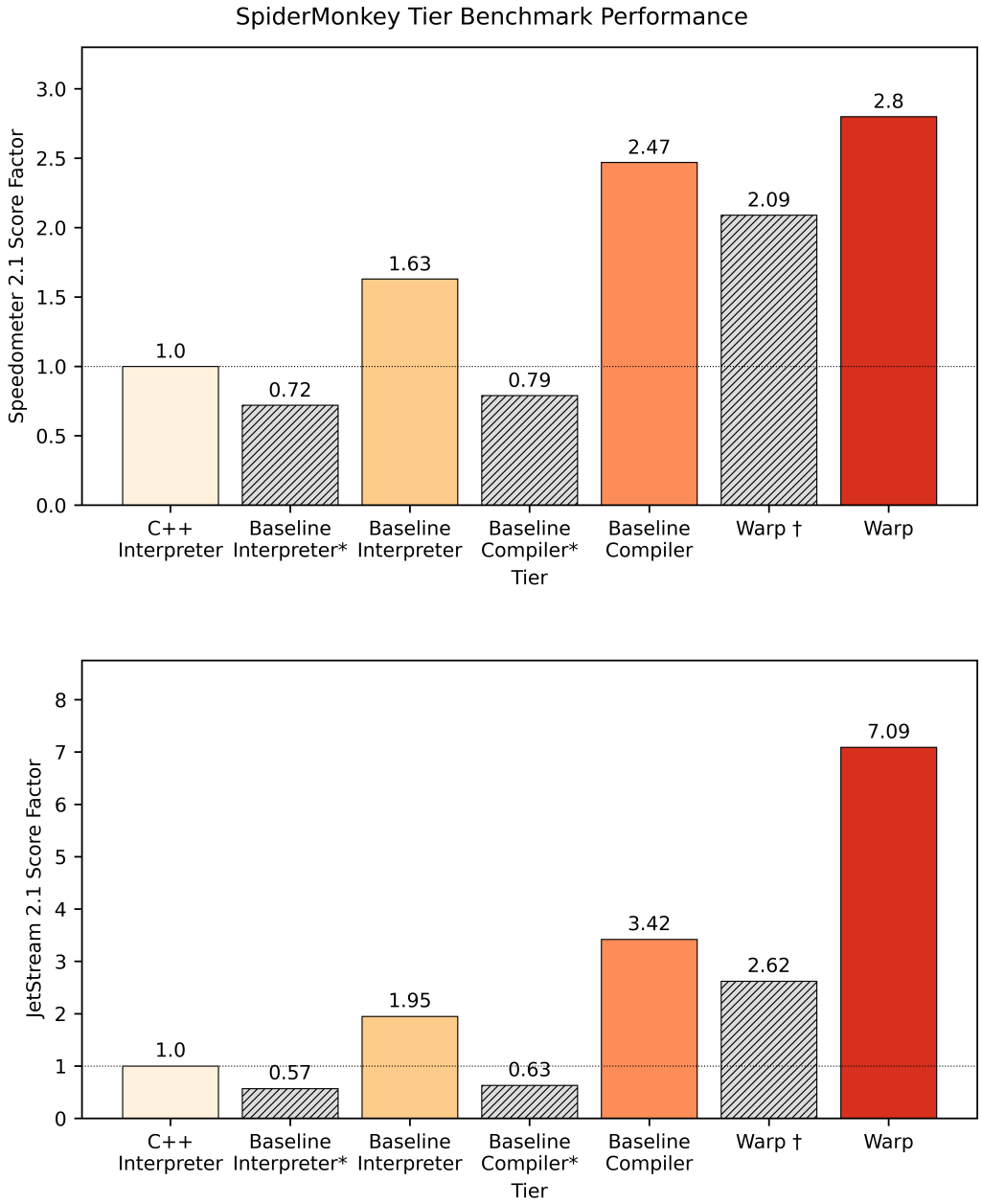

Figure 4.5: Per-tier benchmark score improvement over the ++ Interpreter tier. Tiers marked with a * signify that CacheIR is disabled. Warp† represents disabled CacheIR lowering.

calculated over the lowest tier in the engine, the C++ Interpreter, which is a simple interpreter loop and has no support for inline caching. Enabling higher optimizing tiers leads to better performance for both benchmark suites. At the Baseline Interpreter tier, Speedometer and JetStream observe a 1.63× and 1.95× improvement respectively, directly as a result of the CacheIR system. By exploiting and refining CacheIR information collected in the Baseline Interpreter, the Baseline Compiler further increases performance by 1.5× and 1.75×. Finally, when Warp specializes and optimizes native code by lowering the CacheIR, it increases performance by another 13% and 2×.

To show the value of CacheIR to each tier, Figure 4.5 provides three synthetic tiers where CacheIR's contribution to each tier was removed. The * in the Baseline Interpreter* and Baseline Compiler* tiers indicate that CacheIR is disabled engine-wide. The † represents that only lowering CacheIR in Warp is disabled – in this synthetic tier, Warp compiled function bodies generate inline caches powered by CacheIR but are devoid of type information provided by CacheIR lowering.

The results highlight how much the design of SpiderMonkey leans on the CacheIR system: CacheIR is the most important vector by which performance is gained within the engine. Without inline caching the Baseline Interpreter is strictly overhead as it spends enough time jumping between JIT and C++ code that it is slower than the C++ interpreter in both benchmarks. The Baseline Compiler, which should derive some benefit from the reduction of dispatch overhead shows relatively little benefit over the Baseline Interpreter for very similar reasons.

The disparity between Warp's ability to improve benchmark performance on JetStream and Speedometer reflects that a compiler's ability to improve performance is workload dependent. JetStream has computationally intense kernels which benefit from fine-grained, top-tier optimizations and observes a 2× improvement over the Baseline Compiler. Anh et al. observed that top-tier compilers improve the performance of real-world code much less than computational kernels would suggest [1]. As Speedometer was designed to better reflect the flatter profiles of real-world code, Warp's results on

```
1 GuardNonDoubleType  inputId 0, type Bool
2 LoadOperandResult   inputId 0
3 ReturnFromIC
```

Figure 4.6: First most common CacheIR sequence: Converting a Boolean stored variable to a boolean result.

Speedometer concur with Anh et al.'s observation, providing a more modest speedup of around 13% over Baseline Compiler.

4.3.4 CacheIR: Simplifying IC Development and Enabling Stub Code Sharing

CacheIR makes it easier to develop inline caches for Firefox and can handle a diverse number of cases with relative ease. As a result, running Speedometer and JetStream generates 437 and 531 distinct CacheIR strings respectively, each covering a different case observed in the benchmark run. These are 437 and 531 specialization cases whose CacheIR strings can be forwarded to Warp and lowered throughout the compilation pipeline, massively reducing the complexity of the compiler design required to achieve excellent JavaScript performance.

The diversity of cases covered also helps explain the magnitude of increase provided by CacheIR for Baseline Interpreter performance in Figure 4.5. Since CacheIR models optimized code paths for a large number of diverse cases, it greatly improves performance even without traditional JIT compilation. This is one of the major strengths of the CacheIR design – since covering a new important case with an inline cache is both performant and easy, the SpiderMonkey team has managed to cover a broad range of cases, improving the performance of every tier above the C++ interpreter with one action.

In addition to CacheIR providing a modular IR to share with higher-tier compilers, it also enables native stub code sharing by mapping CacheIR sequences to native stub code. In the Baseline tiers, each CacheIR sequence

Table 4.1: Top ten CacheIR sequences for Baseline ICs with respect to the number of occurrences when running Speedometer. As examples, Figures 4.6 and 4.7 display the two most common sequences. Note: Scripted in this context means a JavaScript function backed by bytecode, as opposed to Native, which would be a JavaScript function backed by C++.

Operation	Occurrences
ToBool(Boolean)	57554
Scripted function dispatch	46264
Load from an object's fixed slot	42629
Load from an object's dynamic slot	28899
Load from a prototype's dynamic slot	26911
Store to an object's fixed slot	15487
Null value check	14933
Integer comparison	12820
ToBool(Object)	12464
Get global name	11139

is compiled to native code once, then each IC stub with the same CacheIR sequence holds a pointer to the compiled code. Table 4.1 illustrates the occurrence count for the top 10 most common CacheIR sequences for Baseline IC stubs in Speedometer. The first and second most common sequences are shown in the listings in Figure 4.6 and Figure 4.7. Native code corresponding to the CacheIR sequences in these listings is shared between $57,554$ and $42,692$ Baseline IC stubs, respectively. By eliminating redundant code compilation for different ICs, CacheIR's design significantly reduces the memory footprint required to support the inline caching system (Section 4.3.5).

4.3.5 WarpBuilder: Exceeding IonBuilder Performance

This section presents a performance evaluation from Firefox 83, the release where WarpBuilder was enabled by default. Firefox 83 is used as a baseline to show what a relatively unoptimized, proof-of-concept version of Warp is capable of delivering. Furthermore, it was the last release where a runtime switch was available to toggle between the old IonBuilder compiler frontend (and associated Type Inference system) and the WarpBuilder frontend with the Type Inference system disabled.

```
1  LoadArgumentDynamicSlot resultId 1, argcId 0,
2                          slotIndex 1
3  GuardToObject           inputId 1
4  GuardSpecificFunction   funId 1,
5                          expectedOffset 0,
6                          nargsAndFlagsOffset 8
7  CallScriptedFunction    calleeId 1, argcId 0,
8                          flags,
9                          argcFixed N
10 ReturnFromIC
```

Figure 4.7: Second most common CacheIR sequence: Calling a specific Scripted function.

Memory Consumption

As part of the design of WarpBuilder, there are two major sources of potential memory savings: (i) removing the need to track and store global type inference to support the TI system in IonBuilder, as discussed in Section 4.2; and (ii) stub code sharing, discussed in Section 4.1.2, meaning that Baseline stubs with identical CacheIR can share native code. Due to the reasons mentioned above, enabling WarpBuilder reduces memory consumption to 0.91× that of IonBuilder for the AWSY tp6 test suite running on Firefox 83.

Benchmark Performance Evolution

Since the release of Firefox 83, further engineering and tuning have greatly improved the WarpBuilder system. To characterize these improvements, this evaluation contrasts the performance results of Firefox 83 and Firefox 111 in Figure 4.8 on Speedometer and JetStream.

For Speedometer running on Firefox 83, enabling WarpBuilder improves the score to 1.15× that of IonBuilder. The improvements on Speedometer are mostly because building Warp around the CacheIR system reduces source code and computation throughout the engine that was previously required to track the global type inference data used by IonBuilder. IonBuilder requires that all functions track type information that is only useful in very hot functions. With

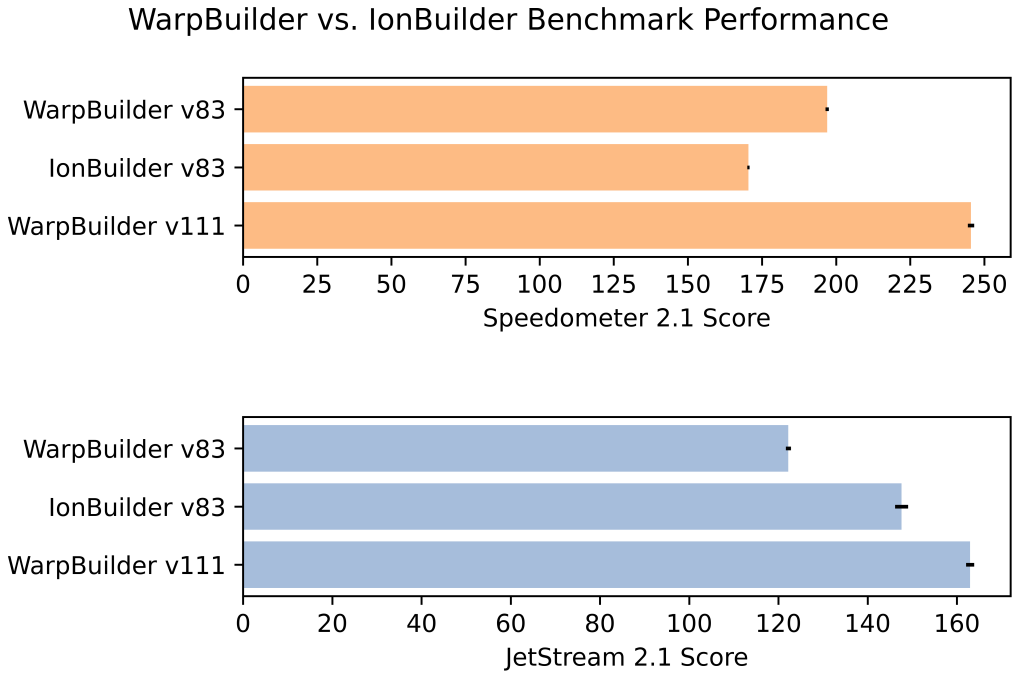

Figure 4.8: Results from running Firefox with WarpBuilder and with IonBuilder for the SPEEDOMETER and JETSTREAM benchmark suites. Each horizontal bar is labeled with the Firefox version number and characterizes the mean score of 15 runs containing the 95% interval as an error. A higher score is better.

WarpBuilder, profiling information via CacheIR is used to both (i) optimize Warp; and (ii) speedup the Baseline Interpreter and Baseline JIT. Warp is also able to do more work off-thread through the Snapshot-based design, allowing the main thread to make progress in program execution while compilation occurs. Moreover, Warp's design is less prone to overspecialization, leading to fewer recompilations than in the IonBuilder system. Building upon WarpBuilder, Firefox 111, which has WarpBuilder enabled by default, achieves a score of 1.44× that of IonBuilder.

For JETSTREAM running on Firefox 83, enabling WarpBuilder reduced the score to 0.83× that of IonBuilder. A substantial fraction of JETSTREAM consists of computational kernel benchmarks which were excellent targets for the combination of Type Inference system and IonBuilder; these types of workloads were intentionally de-prioritized during the development of Warp, and instead focusing on optimizing code that more reflected the state of real

websites. Nevertheless, WarpBuilder's state as of Firefox 111 improves the JetStream score by 1.10× over IonBuilder. WarpBuilder's performance is regained through developing and applying new inline caches via the CacheIR system and expanding the support for compiler optimizations that had yet to be implemented using the CacheIR-based system instead of the Type-Inference-based system.

4.4 Concluding Remarks

This chapter discusses CacheIR, an IC design centered around an IR that simplifies IC development and enables the reuse of compiled native code through IR matching. This chapter also describes and evaluates WarpBuilder, the current JIT compiler front-end used in the SpiderMonkey JavaScript engine that generates specialized code by lowering CacheIR. In SpiderMonkey, the combination of CacheIR and WarpBuilder has proven to be a beneficial design by significantly improving performance across optimization tiers.

Chapter 5

Retaining Type Specialization to Increase the Efficiency of Highly Polymorphic Inline Caches

This chapter addresses the central goal of this thesis and provides an investigation and evaluation of alternative techniques to handle high degrees of polymorphism in operations that use inline caches. The main technique, *Stub Folding*, is motivated by the observation that, in some cases, the fast paths in highly polymorphic ICs share the same code, only differing in the data that they operate on. For highly polymorphic operations, Stub Folding uses code analysis to determine if an IC can retain type-specialized fast-path information by combining all fast paths into one instruction stream and thereby increasing the likelihood of fast-path inlining. Two other techniques investigate the suitability of applying cache replacement policies, such as Least Recently Used to reorder IC fast paths and Least Frequently Used to remove fast paths that are used infrequently.

This chapter's main contributions are the following:

- Stub Folding, a novel approach to handling polymorphic inline caches that consolidates type-specialized fast paths into a unified instruction stream and improves runtime performance by facilitating efficient IC code inlining.
- A Stub Folding implementation in the SpiderMonkey (SM) JavaScript engine and an evaluation compared to SM's previous approach.
- An evaluation of the viability of cache replacement policies for reordering

and eliminating underutilized fast paths in polymorphic inline caches.
An evaluation of Stub Folding in the SpiderMonkey JavaScript engine achieves up to 25% improvement on complex applications within the JetStream 2.1 benchmark suite compared to SpiderMonkey's previous approach. An evaluation indicates that LRU and LFU policies accelerate some programs but do not reliably increase program efficiency across a range of benchmarks.

5.1 Stub Folding

The first technique that this chapter introduces is Stub Folding, an analysis that identifies similarities between cases in a polymorphic inline cache and generates a single code stub. After presenting definitions and the notation used to discuss the analysis (Section 5.1.1), Section 5.1.2 walks through the motivation for the Stub Folding transformation: certain highly polymorphic operation sites that use ICs execute the similar code for each observed type. Then, Section 5.1.3 formalizes the Stub Folding analysis, aimed at identifying stubs that can be folded based on the aforementioned observation, and transformation which folds the identified stubs and updates them when new folding opportunities arise. Lastly, Section 5.1.4 lists the expected benefits, later confirmed through experimental results (Section 5.3.3).

5.1.1 The States of an Inline Cache

A `Stub` is a representation of a type-specialized fast path for a given operation. An `IC` refers to a chain of one or more `Stubs`. `Code` is the native machine code associated with a `Stub`. `Data` encodes `Stub` information — such as object shapes and offsets to their properties — that are accessed by `Code` during execution. `IR` is the intermediate representation that is optimized and used to generate a `Stub`'s `Code`.

There are four states that an `IC` can be in: (i) `Unlinked`: no types have been observed and the `IC` invokes a fallback routine. (ii) `Monomorphic`: a single specialized `Stub` is required effectively handle all observed types. (iii) `Polymorphic`: only up to K `Stubs` are required to effectively handle all observed types.

(iv) `Megamorphic`: greater than K `Stubs` are required to effectively handle all observed types.

5.1.2 A Collection of Stubs as One Logical Stub

In JavaScript, objects have slots that hold values, each associated with some property. To support language features like adding properties to objects dynamically, JavaScript implementations create objects with multiple memory areas for slots. In the first area, there are *fixed slots* that are inlined with the rest of the object metadata. Thus fixed slots contribute directly to the size of objects. As an example, in Figure 5.1 an object `o` is created with two properties `x` and `y`. When allocating space for `o`, the VM provides two fixed or inline slots to the object, speculating that no new properties will be added. Speculating as such can save memory if the speculation is successful, but objects still require a mechanism to add properties dynamically. To do so, there are also *dynamic slots* that are reached through a pointer in the object metadata. Dynamic slots are lazily heap allocated when an object's fixed slots are full and a new property is added. Adding a third property `z` to `o` triggers a memory allocation whose address is held in the object's metadata. The value associated with `z` is stored at this address and any newly added properties are added contiguously after `z` in memory.

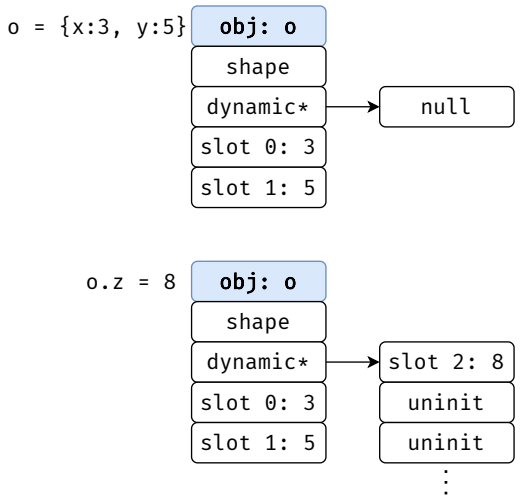

Figure 5.1: Assigning fixed vs. dynamic slots.

```
1  // Different shape, same slot
2  a = {t:1}
3  b = {t:2, u:4}
4  c = {t:8, u:16, v:32}
5
6  // Different shape, different slot
7  d = {x:1}
8  e = {y:2, x:4}
9  f = {z:8, y:16, x:32}
10
11 sum = 0
12 for (o of [a,b,c]) {
13     sum += o.t
14 }
15 for (o of [d,e,f]) {
16     sum += o.x
17 }
```

Figure 5.2: Running example.

From an inline caching perspective, allowing objects to dynamically gain properties has implications in the code that is generated for the same static expression. For a property load, `o.v`, `v` could be stored in both fixed slots and dynamic slots depending on the `Shapes` of the objects represented by `o` at run time. When compiling for a `Stub`, the handler code performs a property lookup, determines if the property is in a fixed or dynamic slot, then generates appropriate `Code` to load the property from the correct location.

For certain `Megamorphic ICs`, the `Code` is equivalent for every `Stub` and only a subset of the fields in `Data` differ. As an example, in Figure 5.2 `o.t` on line 13 and `o.x` on line 16 each access a property from three different objects. Since each object has a different shape, each of these expressions would have an `IC` with three `Stubs`; one for each observed object type. If the `Polymorphic` limit for a number of `Stubs` in an `IC` is two, the VM would employ a slower fallback mechanism to handle all three cases for each expression. However, at each expression, because each property is in a fixed slot for all objects accessed, the `Code` for each `Stub` is equivalent. For `o.t`, the only difference between the fast paths contained in each `Stub` is the object `Shape` held in the `Data`. For

o.x, fast paths differ in both object Shape and property Offset. In such cases, rather than falling back to a slower method to handle megamorphism, it can be more beneficial for the engine to treat the collection of Stubs as one logical Stub, thereby retaining fast path execution and enabling inlining. Stub Folding provides a Code and Data analysis to identify such cases and generate a new Stub accordingly.

In addition to the simplified situations described in Figure 5.2, one of the most common applications of Stub Folding is accessing a shared property stored on a prototype object from a set of distinctly-shaped base objects. These situations are similar to the o.t access in Figure 5.2, where the only difference between the fast paths contained in each Stub is the object Shape held in the Data, but instead of accessing the property directly from the base object, it is accessed from the prototype.

5.1.3 Identifying Opportunities and Verifying Legality Conditions

Stub Folding requires that the code executed be equivalent and that the data manipulated be similar. This section explains: (i) when and how an IC invokes the analysis required to create a FoldedStub; and (ii) the congruence analysis required for updating an existing FoldedStub to handle a new case.

Code Equivalence and Data Similarity

If an IC receives an incoming case that cannot be handled appropriately by any existing Stub, it invokes a fallback mechanism. When this occurs from a Megamorphic-candidate IC, the VM calls a routine to identify applicable cases for Stub Folding.

This routine takes the Megamorphic-candidate IC as input. As a running example, this section uses line 16 from Figure 5.2 with a Polymorphic limit of two Stubs before transitioning to Megamorphic. Two conditions need to be met for the analysis to consider legal to fold the given IC:

1. All Stubs have the same Code sequence.
2. All Stubs have the same Data, except for (i) a single Shape used in a

`Shape Guard` and (ii) for property loads or stores, a subsequent `Offset` value specifying the property offset from the beginning of the containing memory region.

To simplify testing `Code` equivalence across `Stubs`, equivalence is determined based on the associated `IR` rather than the native machine code directly. The analysis verifies the first and second conditions by maintaining a copy of the first `Stub` and iterating over all `Stubs` in the `IC`. The beginning of each iteration of the analysis determines `Code` equivalence between `Stubs` by comparing the `IR` of the first `Stub` — $\texttt{IR}_0$ — with the `IR` of the subsequent `Stubs`. For $\texttt{Stub}_i$ processed in the i^{th} iteration, if $\texttt{IR}_i$ fails the equivalence check, the analysis exits early and does not apply the transformation. Once $\texttt{IR}_i$ equivalence is verified, the analysis walks through $\texttt{Data}_i$ to ensure that condition 2 holds. Initially, the goal is to verify that there are at most two data fields — a `Shape` and an `Offset` — in the $\texttt{Data}_i$ that differ from $\texttt{Data}_0$. The first differing field must be of the `Shape` type. If any field differs that is not of type `Shape`, then the analysis exits early and does not apply the transformation. Once the analysis identifies a differing `Shape`, it saves the `Shape` value into a `ShapesList` and records its *location* within `Data` (`ShapeLocation`). Figure 5.3 shows the representation of the `IR` and `Data` for each `Stub`. In this case, the analysis determines that the `IR` is consistent across all `Stubs`, verifying the first condition. The first differing field between the `Data` is a `Shape` and thus `ShapesList` contains the shapes for `f`, `e`, and `d`, and `ShapeLocation` is `0`.

The remaining fields in $\texttt{Data}_i$ must be equivalent to $\texttt{Data}_0$ except for, potentially, an `Offset` value that corresponds to the offset required to access a property for a given `Shape`. The analysis locates and saves the location of `Offset` within `Data` — `OffsetsLocation`. Once identified, the analysis begins collecting all of the `Offset` values into an `OffsetsList`. For Figure 5.3, the analysis collects the `Offsets` `2`, `1`, and `0` into `OffsetsList` and sets `OffsetsLocation` to `1`.

The analysis walks through the `IR`, mapping `IR` operations with the fields in `Data` that they operate on. The analysis exits early and does not apply the transformation in two situations: (1) `Data[ShapeLocation]` is not used

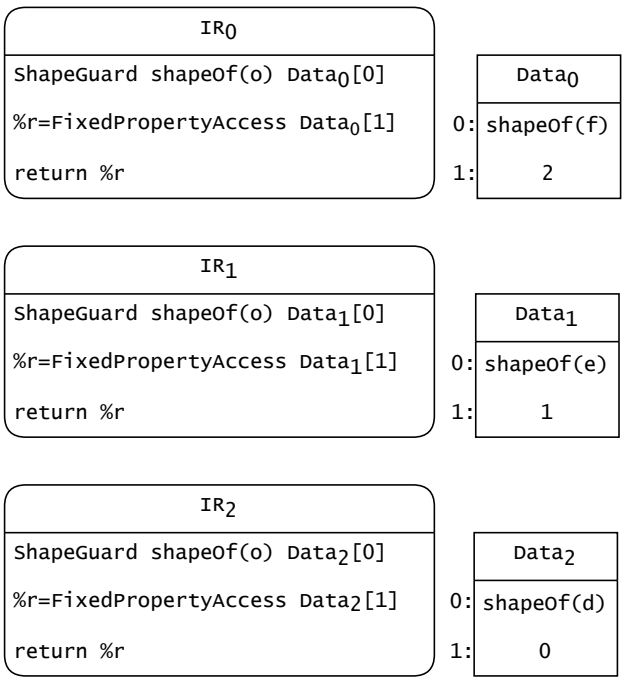

Figure 5.3: Running example (line 16 of Figure 5.2) before Stub Folding.

by a `ShapeGuard` operation; or (2) `Data[OffsetsLocation]` is not used by a `PropertyAccess` operation In Figure 5.3, `Data[0]` is used by a `ShapeGuard` and `Data[1]` is used by a `FixedPropertyAccess`, thus the analysis verifies the second condition.

When both conditions are met, the generation of a new `FoldedStub` creates a new `IR` by copying and modifying the previous `IR`. The identified `ShapeGuard` operation is replaced with a `MultipleShapeGuard` operation. If the `Offsets` in `OffsetsList` are not all identical, the associated `PropertyAccess` operation is replaced with a `MultiplePropertyAccess` operation. Similarly, `Data` is copied and modified to create the `FoldedStub`'s `Data`. At the `ShapeLocation`, a pointer to the `ShapesList` replaces a single `Shape` value. If the `Offsets` in `OffsetsList` are not all identical, a pointer to the `OffsetsList` replaces a single `Offset` value at the `OffsetsLocation`. The compiler uses `IR` to generate `Code` that operates on the new `Data`. Finally, the single `FoldedStub` replaces all of the `Stubs` in the `IC`. Figure 5.4 shows the `FoldedStub` for the running example. The `MultipleShapeGuard` operation now guards within a list of `Shapes`. The operation compares the incoming `Shape` with each `Shape` in

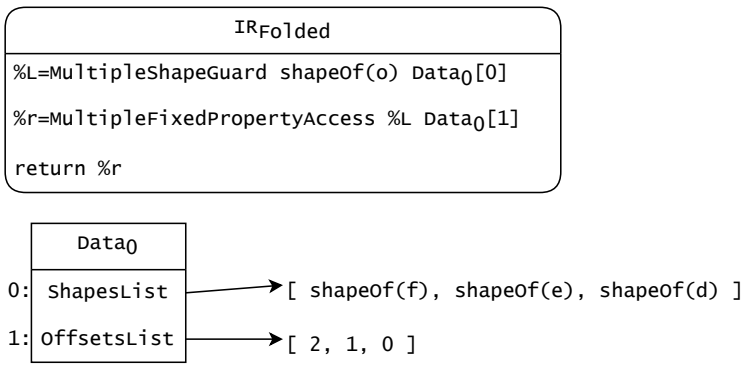

Figure 5.4: Running example (line 16 of Figure 5.2) after Stub Folding.

the `ShapesList`. If the operation finds a match, it returns the index of the matching element. The `MultipleFixedPropertyAccess` operation consumes the index to retrieve the corresponding `Offset` from `OffsetsList`. With the correct `Offset`, the `FoldedStub` retrieves and returns the appropriate property value.

Congruence Analysis and Folded-Stub Updating

When an `IC` with a `FoldedStub` invokes a fallback mechanism because all the `Stubs` fail to handle an incoming case, the VM calls a routine to determine if a new case could be added to the `FoldedStub`.

Given an `IC` with a single `FoldedStub` and the `IR` and `Data` of the unhandled case, the analysis walks through the `FoldedStub`'s `IR` and the unhandled case's `IR` simultaneously, verifying congruence in IR operations. Two IRs are congruent if all IR operations, other than the `MultipleShapeGuard` and `MultiplePropertyAccess`, are equivalent. These operations are only considered congruent, respectively, to a `ShapeGuard` and the appropriate kind of `PropertyAccess` operations (i.e. an access from either a fixed slot or from a dynamic slot) from an unhandled case's IR. When there is full congruence, the analysis appends the unhandled case's `Shape` to the `FoldedStub`'s `ShapesList` and, if there is a `MultiplePropertyAccess`, the `Offset` to the `OffsetsList`. Adding support for a new case in a `FoldedStub` does not require compiling the `Code`.

If the IR operations are not congruent, the `FoldedStub` is not updated and the VM instead creates a new `Stub` and places it at the beginning of the `IC`.

5.1.4 Stub-Folding Potential Benefits

There are three potential benefits of applying Stub Folding: (i) the number of `Stubs` permitted in an `IC` before it transitions into a `Megamorphic` state is often arbitrarily chosen, and applied to all `IC`s — Stub Folding provides continued fast-path support at operation sites that are fixed-case polymorphic but exceed that arbitrary limit; (ii) Stub Folding retains data and `IC` information local to an `IC` to reduce pressure on the alternative fallback methods, which are often a global cache lookup for property loads that suffer from frequent evictions; (iii) Stub Folding `Polymorphic` or `Megamorphic ICs` to be treated as a `Monomorphic IC` allowing, in a tiered execution environment, for greater optimizations in the JIT pipeline such as inlining `Code` into an optimized function body. The `Code` of `Monomorphic ICs` is often inlined into the function due to speculative type stability. Inlining `Polymorphic ICs` is less common because they contain multiple distinct code sequences. Adding a new case to such `ICs` would require the function to be recompiled to account for the newly added case. Stub Folding allows the compiler to inline a single code sequence into the hot function, providing fast-path execution for a larger number of cases. Moreover, since `FoldedStubs` can be updated without recompilation, adding a previously unhandled case to a `FoldedStub` maintains the validity of the compiled function.

There are, however, cases where a function with a `FoldedStub` would need to be recompiled. If, for example, the congruence analysis determines that an unhandled case is not congruent with the `FoldedStub`, the function would need to bail out to a lower tier so that it can be recompiled without the `FoldedStub`. In such cases, an alternative fallback mechanism, such as the global cache lookup mentioned previously, may be more appropriate.

5.2 Strategies Inspired by Cache-Replacement Policies

This chapter also explores two alternative policies for handling polymorphic inline caching that are inspired by the replacement policies found in hardware and software caches.

5.2.1 Least Recently Used Policy

The problem of handling many stubs in an `IC` is analogous to the problem of having too many entries that must be kept in a set in a hardware cache. When capacity is limited, a replacement policy is needed. Least-Recently Used is a classic replacement policy that keeps track of the recency of access to each of the entries currently in the cache and, when one must be replaced, it selects the one that has not been recently referenced for eviction. When a lookup into the cache is performed sequentially, a design variation may use the recency of access to order the lookup such that the ones referenced recently are checked first.

LRU in `ICs`

One way to apply these ideas to an `IC` design is reordering the `Stub`s in the `IC` each time that a `Stub` is used. The reordering ensures that the most recently used `Stub` is always checked first and the search through the `IC` follows the recency of use. This technique may incur an overhead of reordering `Stub`s in each access to the `IC`, thus an evaluation of this idea must measure the benefit of finding a `Stub` sooner versus the cost of `Stub` reordering. When executing a bytecode with an `IC`, the technique records which `Stub` successfully handled the incoming case and moves the `Stub` to the front of the `IC`. The next time the program executes that bytecode, the most recently successful `Stub` is checked first. This technique maintains a recency ordering – the recency property – for the `IC` based on cache hits. LRU can also be used as an actual replacement policy in an `IC` design: when an `IC` is about to transition to a `Megamorphic` state: (i) remove the least recently used `Stub` (the last `Stub` in the `IC`), and

(ii) keep the IC in a Polymorphic state.

Based on this idea two techniques are investigated:

LRU Eviction: Combines recency ordering with the IC LRU replacement policy. The IC never transitions to a Megamorphic state and always retains at most k Stubs.

LRU No Eviction Applies only recency ordering. The IC may transition to a Megamorphic state but while in Polymorphic state it maintains the recency property of the IC.

5.2.2 Removing Inactive Stubs

This technique is a Stub-activity-based analysis that targets reducing the number of unnecessary Megamorphic ICs. The analysis aims to detect and react to phase changes in the program, which can affect the set of objects seen at an operation site. Removing Inactive Stubs is inspired by the cache Least Frequently Used policies.

LFU in ICs

This analysis is run on a Megamorphic-candidate IC, one which has the limit of Polymorphic cases but hits the fallback routine. The algorithm is as follows:

1. For each Stub in a Megamorphic-candidate IC, calculate the hit count, hc.
2. If the hc < k, where k is a tunable parameter, remove the Stub from the IC.
3. If the resulting IC has greater than m Stubs, transition to a Megamorphic state, otherwise, remain in a Polymorphic state.

By removing Stubs that have a hc below the threshold, this analysis aims to preserve the type-specialized fast paths for the most frequently executed cases. This analysis ensures that the set of Stubs in an IC adapts to the current set of types that an operation site observes. In the evaluation for Removing Incative Stubs, $k = 1$ and $m = 4$.

5.3 Evaluation: Stub Folding Improves Performance, Other Techniques' Results are Mixed

This subsection evaluates Stub Folding, LRU Eviction, LRU No Eviction, and Removing Inactive `Stub`s. Results indicate that: (i) Stub Folding accelerates a diverse collection of programs, (ii) LRU- and LFU-based policies do not reliably increase program efficiency and can lead to drastic degradation.

5.3.1 Benchmarks, Hardware, and Software

This subsection presents data collected from the Speedometer 2.1 [29] and the JetStream 2.1 [18] benchmark suites running on a machine equipped with an Intel i5 12400 with 16 GiB of DDR4 memory that runs Fedora 36 (Kernel 5.18.11-200). Speedometer and JetStream are major benchmark suites frequently used to evaluate modern browsers' JavaScript engines such as SpiderMonkey, JavaScriptCore, and V8.

Speedometer 2.1 contains 16 subtests, all of which are `To-Do list` applications written in different popular JavaScript frameworks such as React, React with Redux, Ember.js, Backbone.js, AngularJS, Vue.js, jQuery, Preact, Inferno, and Flight. These frameworks are ubiquitously used in Web development and Speedometer is meant to reflect real-world Web-App workloads by conducting operations on a `To-Do list` — adding items, deleting items, and completing items — both synchronously and asynchronously. Speedometer data points are the median run time in milliseconds across all iterations.

JetStream 2.1 contains 64 subtests evaluating computationally intensive workloads.[1] On each invocation of JetStream, each subtest runs for 120 iterations and computes a score value:

$$subScore = \frac{5000}{mean(time)} \tag{5.1}$$

where $mean(time)$ is the mean time, measured in milliseconds, taken across all iterations of a subtest; and the 5000 is a *magic*[2] number used to normalize the

[1] A full list is found here: https://browserbench.org/JetStream2.1/in-depth.html
[2] This decision was made by the authors of JetStream 2.1.

resulting score value to ensure higher scores correspond to faster run time.

5.3.2 Experimental Methodology

The results presented in Sections 5.3.3, 5.3.4, and 5.3.5 use Mozilla's `mach` tool with the `raptor` subcommand to collect benchmark metrics for Speedometer and JetStream. For baseline measurements, each evaluation uses a version of Firefox 109.0.1[3]. Each technique is implemented in a child patch from the baseline revision.

Each data point in the figures represents the metrics described above computed from 15 `mach` invocations for both Speedometer and JetStream. For JetStream, each of the 15 invocations produces 120 iterations, therefore, the data points are calculated by taking the arithmetic mean across all iterations – 15×120 – then using Equation 5.1 to compute the final score. In all figures, higher percentages correspond to performance improvement over baseline for the evaluated method. The ordering of subtests on the x-axis of the figures differ from one another. Subtests are ordered from lowest performing to highest performing for each evaluated technique. The figures only include subtests whose baseline metric and evaluated technique metric have non-overlapping 95% confidence intervals. Note, that the error bars are omitted from the graphs because the variation is minimal. Lastly, hardware counters were collected with the Linux `perf` tool.

5.3.3 Stub Folding Evaluation

Figure 5.5 and Figure 5.6 present the effects of Stub Folding on JetStream and Speedometer, respectively. For JetStream, the subtests that benefit the most from Stub Folding are parser applications. `Babel`, `babel-wtb`, and `acorn-wtb` are parsers for JavaScript and rely heavily on polymorphic object hierarchies. Stub Folding excels in these cases because an operation may observe many types that access a property from a common prototype object. Stub Folding recognizes that the code generated to access the prototype's property

[3]Revision bc4345d1f736.

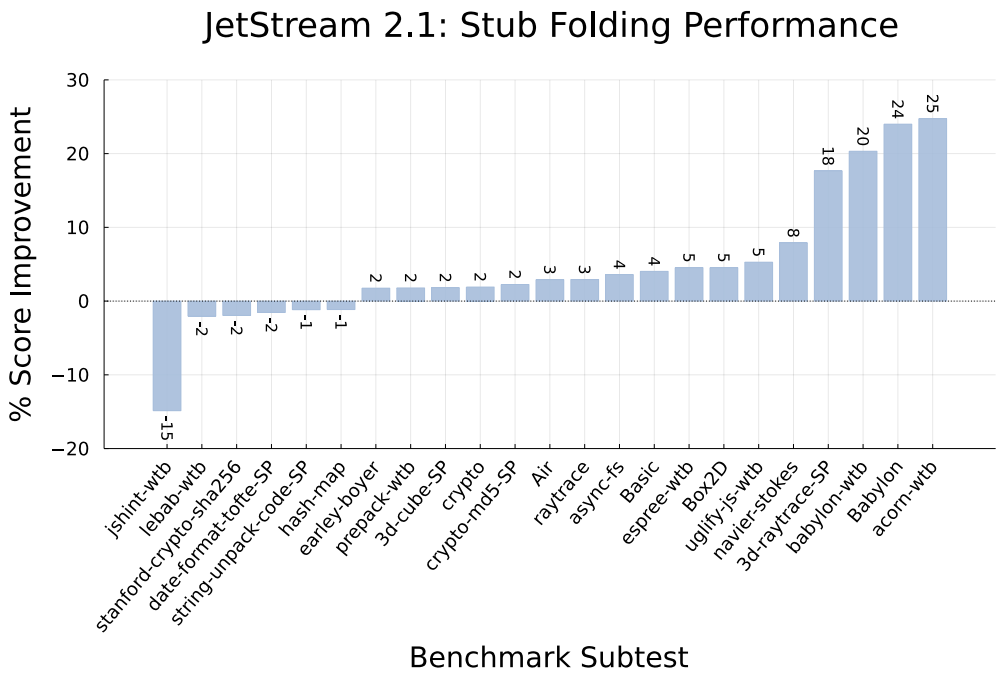

Figure 5.5: Percentage Score improvement for Stub Folding over baseline on the JetStream 2.1 subtests.

is the same even when the types of the direct objects are different. The performance improvements in the benchmarks stem from Stub Folding allowing lower execution tiers to retain fast-path information and higher execution tiers to inline `FoldedStub Code` into optimized function bodies. As an example, Stub Folding is applied to 1064 `IC`s in `acorn-wtb`. Out of those `IC`s, only 53 fail to update the `FoldedStub` and continue to attach distinct `Stub`s. For the remaining 1011 `IC`s, the `Code` from the `FoldedStub`s are successfully inlined when the JIT compiles entire functions. This means that, for `acorn-wtb`, roughly 95% of the Stub Folding opportunities successfully contributed to improving the performance of compiled code. These improvements reduce the pressure on the global cache used for `Megamorphic` property accesses, leading to faster lookups for a greater number of `Megamorphic` accesses in the program.

The performance degradation in `jshint-wtb` is attributed to a lower success rate for Stub Folding. In `jshint-wtb`, only roughly 66% of the Stub Folding opportunities are eventually inlined. Without a high success rate, Stub Folding's analysis introduces too much overhead to identify Stub Folding cases and create

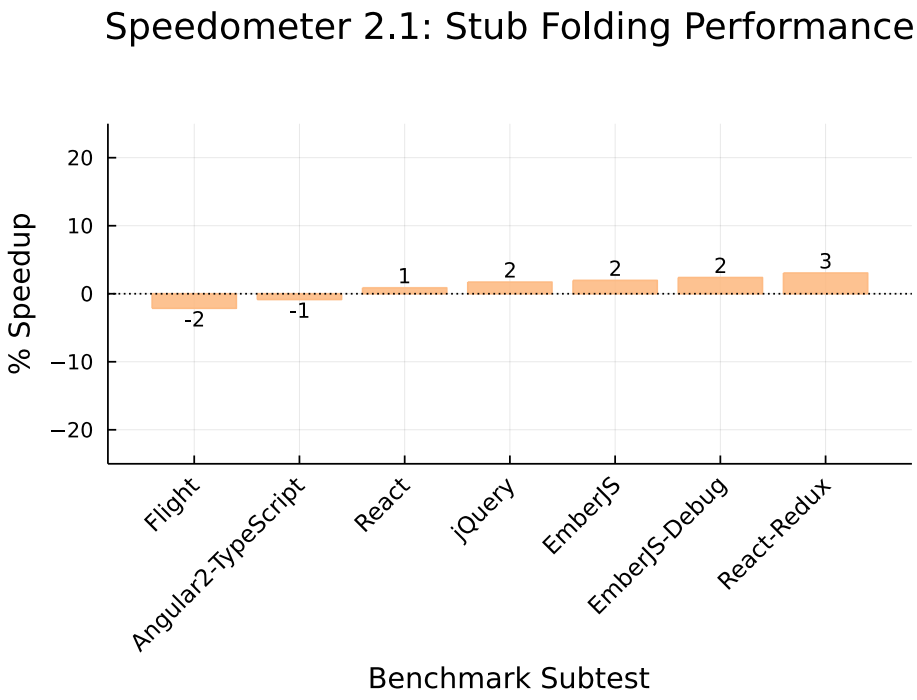

Figure 5.6: Percentage speedup for Stub Folding over baseline on the Speedometer 2.1 subtests.

the `FoldedStub`. Despite this degradation, the large performance improvements in the other subtests, and the moderate performance improvements for the majority of the subtest in Figure 5.5, benefit the overall benchmark performance.

For Speedometer, as seen in Figure 5.6, Stub Folding had a milder impact on subtest performance. However, Stub Folding delivers a modest overall performance improvement due to the subtests that improve by 1% to 3%.

5.3.4 LRU Policy Evaluation

Figure 5.7 and Figure 5.8 present the effects of LRU Eviction on JetStream and Speedometer, respectively. These results indicate that maintaining a fix-size maximum number of `Stubs` for an `IC` and preventing all `Megamorphic ICs` from adopting a generic routine to compute the result can severely degrade performance. This degradation is attributed to the overhead of maintaining the recency ordering for `Megamorphic ICs` and the low hit rate on the reordered `Stubs`. For example, `jshint-in` in Figure 5.7 degrades performance to 37% compared to the baseline. Maintaining recency ordering for `Stubs` with low hit

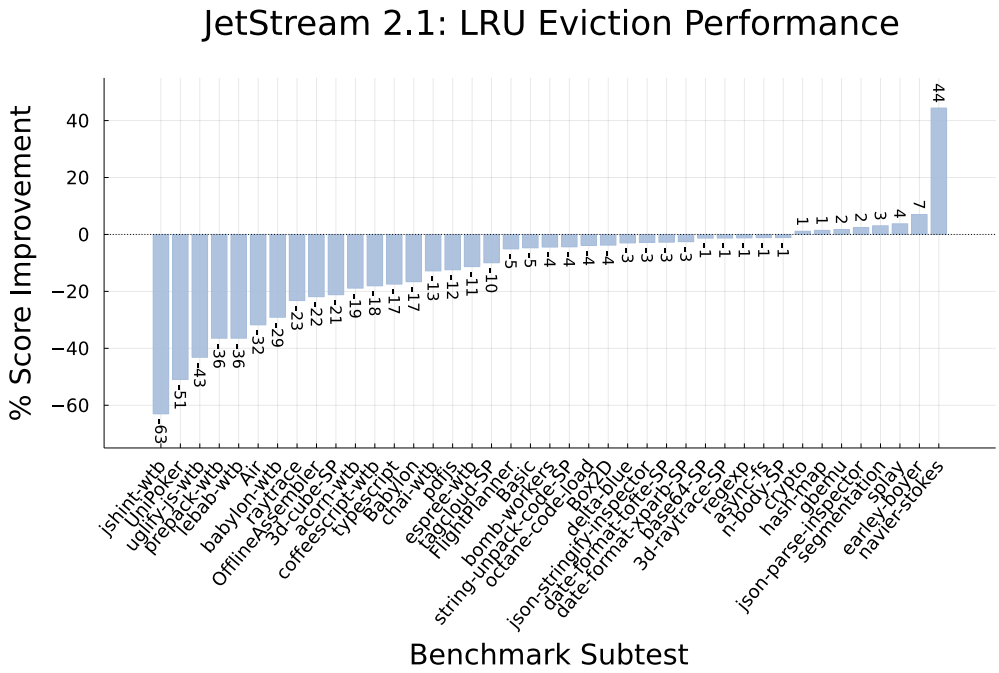

Figure 5.7: Percentage Score improvement for LRU Eviction over baseline on the JetStream 2.1 subtests.

rates leads to roughly (i) 2× the number of cycles, number of instructions, and number of branches; (ii) 2.3× the number of branch misses; and (iii) 3× the number of L1, L2, and L3 cache misses. An increase in the number of dynamic instructions is a consequence of preserving the recency ordering, as running the reordering logic becomes necessary when the most recently used `Stub` is not hit. Additionally, not hitting the most recently used `Stub` also leads to an increase in dynamic branches and branch misses.

Figure 5.9 and Figure 5.10 present the effects of LRU No Eviction on JetStream and Speedometer, respectively. By maintaining the recency property of an `IC`, but allowing a more generic fallback mechanism for highly `Megamorphic ICs`, the approach can minimize performance degradation in pathological cases while maintaining the performance benefits in successful cases. Figure 5.9 indicates a successful application of the LRU No Eviction to the `navier-stokes` subtest. This result suggests that `navier-stokes` has operation sites that are `Polymorphic`, but not `Megamorphic`, wherein fast paths represented by `Stubs` have long periods of hits before another `Stub` is needed to correctly handle the

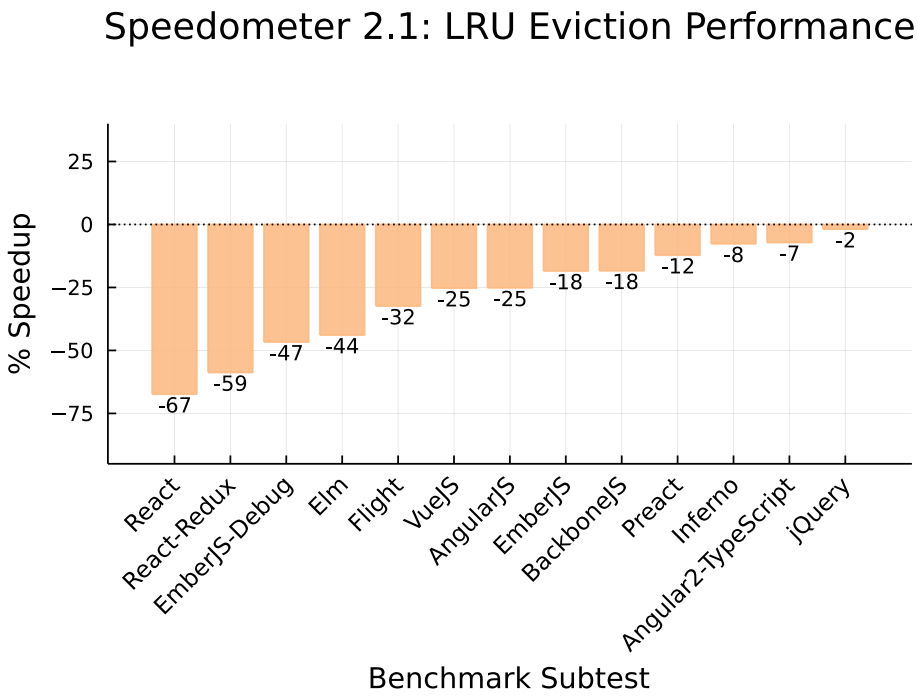

Figure 5.8: Percentage speedup for LRU Eviction over baseline on the Speedometer 2.1 subtests.

incoming case. LRU No Eviction applied to `navier-stokes` reduces hardware counters to roughly (i) 0.5× the number of cycles, number of instructions, and number of branches; (ii) 0.1× the number of branch, L1, and L2 misses; and (iii) 0.3× the number of L3 misses. The reduced number of branches and branch misses reflects that the recency ordering is successful at increasing the `Stub` hit rate and reducing the number of failed guard conditions needed to find the appropriate `Stub` to handle the incoming case.

The Speedometer results in Figure 5.10 indicate that even though LRU No Eviction reduces the negative effects of LRU Eviction, the approach is not effective in increasing the efficiency for any of the subtests.

5.3.5 Evaluating the Removal of Inactive Stubs

Figure 5.11 and Figure 5.12 present the effects of applying the `Stub`-activity-based analysis from Section 5.2.2 on JetStream and Speedometer, respectively. Though a small number of subtests have statistically significant improvements over the baseline, the overall benchmark scores are degraded for both bench-

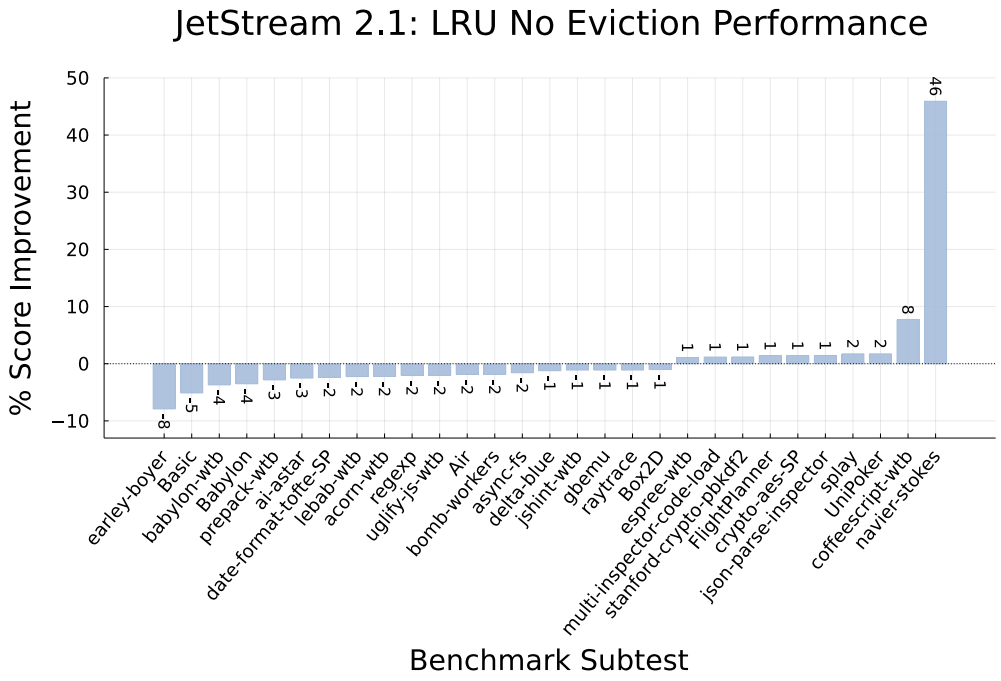

Figure 5.9: Percentage Score improvement for LRU No Eviction over baseline on the JetStream 2.1 subtests.

marks. There are two sources of overhead causing degradation: (i) For `ICs` that have no active `Stubs`, the analysis is entirely overhead. (ii) For `ICs` that frequently attach many `Stubs` — meaning that the set of objects seen at an operation site frequently changes — the analysis overhead outweighs the benefits of retaining fast paths for cases that are infrequently seen later in the program. Removing Inactive `Stubs` excels on code with infrequent shifts in sets of fast paths required at an operation site. Implementing the analysis in SpiderMonkey and evaluating it on JetStream and Speedometer indicate that an analysis that evicts `Stubs` from an `IC` based on hit-count frequency is not a generally successful approach to increase program efficiency.

5.4 Concluding Remarks

This chapter presents Stub Folding, a novel approach to handling polymorphic inline caches based on fast-path-code equivalence, and an evaluation of Stub Folding comparing it to SpiderMonkey's previous inline caching scheme. Results from this paper indicate that Stub Folding has merit as an alternative or

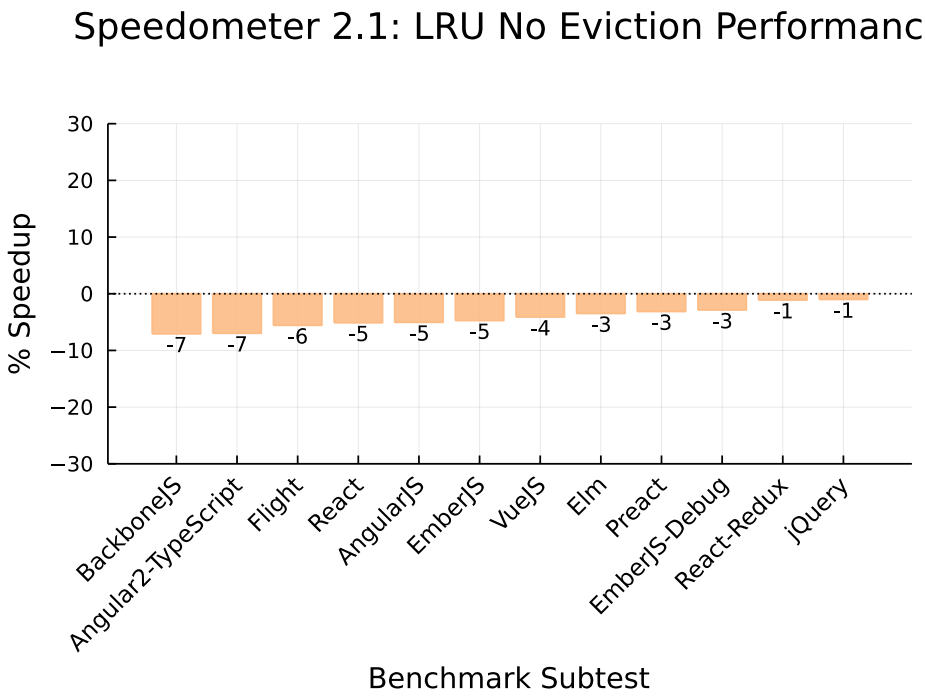

Figure 5.10: Percentage speedup for LRU No Eviction over baseline on the Speedometer 2.1 subtests.

cooperative approach to handle polymorphic inline caches in dynamically-typed language implementations. Additionally, this chapter also evaluates two strategies inspired by hardware caching policies. An evaluation of two variations of a Least Recently Used approach shows that reordering inline cache fast paths based on recency is highly successful in certain cases, but can lead to degradation if not handled correctly. Moreover, an analysis that evicts inline-cache fast paths based on a hit count frequency does not reliably increase program efficiency for the evaluated benchmark programs.

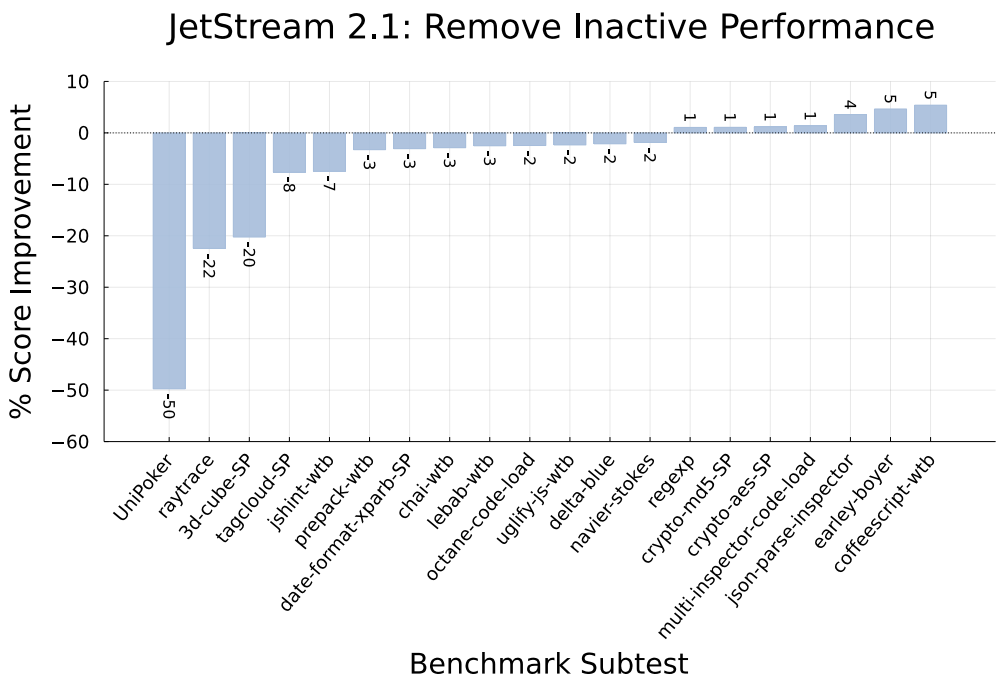

Figure 5.11: Percentage Score improvement for Remove Inactive Stubs over baseline on the JetStream 2.1 subtests.

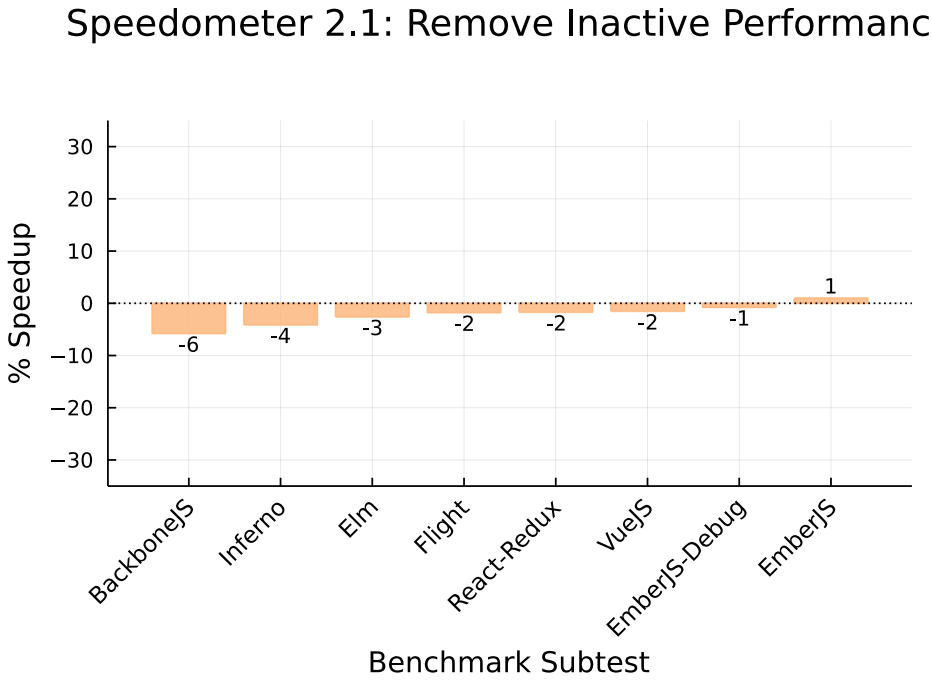

Figure 5.12: Percentage Speedup for Remove Inactive Stubs over baseline on the Speedometer 2.1 subtests.

Chapter 6

Related Work

6.1 A Unifying Abstraction for Inline Caches

According to Deutsch and Schiffman, the key result behind their Smalltalk-80 implementation is *Dynamic Change of Representation* [9]. Under this rubric, JIT compilation (dynamic translation in their terminology) and Inline Caching are different representations of method dispatch. They introduce a one-entry IC, generated by their JIT compiler, that initially *unlinked* and is then patched with generated native code containing an updated target. They also discuss inlining small methods for commonly used selectors such as `+`. In a sense, the generated CacheIR for a particular bytecode is an inlined and type-specialized version of a selector, a *dynamic change of representation* similar to Deutsch and Schiffman's system. However, Deutsch and Schiffman do not describe a unifying mechanism underlying their inline caching. Thus, in their design, optimizations are implemented by hand.

Polymorphic Inline Caches (PICs) were first described in the context of the SELF language by Hölzle et al. [15]. The main idea of their PIC scheme is to provide a mechanism wherein, for polymorphic call sites, instead of overwriting the call site directly with the address of a single resolved method, a machine-code stub is created, and the address of the stub is used inline. Their work is foundational for PIC support in modern language systems; the underlying system outlined in this paper is a clear reflection of the ideas presented by Hölzle et al. However, unlike CacheIR, Hölzle et al. scheme does not rely on a specialized IR to create IC stubs. The presence and architecture of CacheIR in

SpiderMonkey enables high levels of native stub code sharing that is contingent on an equality check of the IR itself. This code sharing leads to less frequent stub compilation and greater memory efficiency to drive inline caches.

In later work, within the context of the SELF language, Hölzle et al. describe the use of ICs and PICs to drive compilation decisions in the runtime system [16]. The insight is that the SELF PIC scheme could be used as a type-information collector in addition to accelerating operation sites. By leveraging this type information, they speculatively compile hot functions with potentially stabilized types. CacheIR builds greatly upon this insight and enables IC stub code structure to be reached throughout the entire compilation pipeline. CacheIR is used as a shared source of truth in the engine, having prevalent use from the Baseline Interpreter through to the optimizing Ion compiler providing an efficient means for stub-code lowering.

LIL [12] is an architecture-independent language for writing Virtual Machine stubs. Compared to CacheIR, LIL has a much broader set of responsibilities — it covers many assembly-code stub responsibilities, such as object allocation and garbage-collector barriers — and thus supports arithmetic operations, conditional control flow, multiple calling conventions and more. LIL is a mechanism for replacing assembly-written stubs, but it does not appear that LIL is tightly connected to their IC infrastructure.

The linear intermediate representation CacheIR uses is akin to a *trace*, from a tracing JIT compiler, such as TraceMonkey or PyPy [4], [11]. Traces have very similar side exit mechanisms, also called *guards* where a precondition is required to proceed further into the trace. However, unlike in a tracing JIT the failure of a guard in CacheIR simply starts execution again at the beginning of the next stub. Unlike a fully developed tracing JIT, CacheIR is constructed incrementally by program developers rather than derived directly from execution. This is because CacheIR operates at a sub-bytecode granularity, and as such there is no infrastructure to collect traces automatically.

Wuthinger et al. [35] introduce Truffle, a framework that provides a modular, unified infrastructure and architecture for building dynamic languages VMs. Their architecture enables node rewriting on an AST to specialize execution at

runtime. To enable specialization, Truffle supports polymorphic inline caching by expanding a node representing an IC-supported operation to multiple nodes each specialized to an observed type. Similar to IC stubs in SpiderMonkey, Truffle's IC nodes are chained, connected by an edge representing the failure path for the preceding node. However, the IC scheme in Truffle is more akin to Quickening [5]; rather than backing an IC with dynamically compiled stub code, a generic VM call is replaced with a specialized VM call. Thus, unlike WarpBuilder and CacheIR, no abstraction within Truffle and Graal enables native code sharing between ICs. An independently developed idea similar to Trial Inlining, called *Polyvariance*, exists in the JSC JavaScript engine powering WebKit [28]. The goal of both Trial Inlining and Polyvariance is to use ICs to specialize polymorphic operations on a per-call-site basis. Polyvariance works by inlining small functions when tiering up from the JCS Baseline compiler to the next-tier DFG compiler. These inlined functions have unlinked ICs associated with DFG. If DFG detects that an IC fast path is taken through the inlined function, it fills in the unlinked ICs, thus enabling more specialized code generation in the next, highest-tier compiler. Polyvariance differs from Trial Inlining in a few notable ways. First, Trial Inlining is an intermediate step between Baseline and Warp whereas Polyvariance is applied later in JSC's JIT pipeline. Thus, SpiderMonkey's Baseline compiler can detect some of the benefits of Trial Inlining and preemptively collect more precise profiling data before tiering up. Second, due to SpiderMonkey's IC architecture, Trial Inlining specializes in moderately deep nestings of call sites whereas Polyvariance is a one-level-deep inlining strategy. Third, Trial Inlining does not inline, in the sense of function inlining, until tiering up to Warp. Instead, Trial Inlining creates a specialized set of ICs for a call site, leaving the actual function inlining decision to Warp.

CacheIR has further utility not discussed in this thesis. *Cachet* is a domain-specific language intended to help build a formally verified JIT compiler [27]. In their prototype implementation, they build a CacheIR compiler for Cachet to help on the road to formal verification for the SpiderMonkey JIT compilers.

6.2 Techniques for Highly Polymorphic Inline Caches

In the seminal PIC paper, Hölzle et al. [15] select ten as the maximum number of type-specialized fast paths for an operation site, before considering the site to be megamorphic. Ten were selected based on the rarity of exceeding ten types of objects at an operation site in the evaluated SELF programs. However, their work suggests a few future directions for more sophisticated mechanisms to manage polymorphism in inline caches. Based on these suggestions, the present study implements and evaluates an analysis based on hit frequency for removing fast paths that are no longer active. Other suggestions like improving linear search to find the correct fast path implementation remain unstudied.

Ahn et al. [1] identify the challenges that JavaScript implementations face when dealing with types and propose a type system that decouples the prototype pointer from an object's type. They posit that type stability is hindered by including the pointer to the object's prototype in the object's shape encoding. While their experiments show a reduction in execution time, removing the prototype pointer from shapes eliminates the opportunity for other optimizations that rely on the shape implying the prototype.

Serrano et al. [24] extend a polymorphic inline caching strategy with a technique based on virtual tables (vtables) from statically typed language implementations like C++. Their technique builds a shape-associated vtable at runtime. Each vtable contains an entry for every property access site in the program and each entry contains the offset needed to access the property for that shape. Because objects of each shape are unlikely to be seen at each operation site, the vtables can be very sparse, consuming unnecessary memory. Shape data structures are altered to contain a pointer to the associated vtable, using more memory to encode each shape. Unlike Stub Folding, Ahn et al.'s and Serrano et al.'s work requires fundamentally altering shape encoding.

Another work by Serrano et al. [25], extends an inline caching scheme to accelerate property accesses with dynamic property names and thereby accelerate property accesses on proxy objects. To accelerate property accesses

with dynamic property names, they propose adding a cache to string values encoding the names rather than an inline cache associated with the access operation. Strings are altered to contain a cache that saves the object shape whose property is accessed by the name and the property slot index needed to access the property. Proxy objects in JavaScript do not have properties themselves but can define traps on reads and writes to another object. Proxy traps generally access properties of proxied objects with dynamic property names and the previous solution accelerates these cases. Again, this technique relies on increasing the size of primitive string encoding. Moreover, proxy objects are used relatively rarely in real-world workloads, and their solutions do not accelerate property accesses more generally.

To reduce JavaScript start-up time, Choi et al. [8] present a system to reuse ICs between program invocations. Reusable inline caching (RIC) links the objects used at property access sites from an initial program run to subsequent runs and pre-populates the access sites with appropriate handler code. RIC reduces library start-up time by 17% on average by avoiding initial IC misses on subsequent execution of library code. Unlike Stub Folding, RIC does not explicitly propose techniques targeting highly polymorphic access sites.

Milojković et al. [19] propose inline-cache type inference (ICTI), a heuristic approach to improve the precision of static type inference systems with dynamic feedback from inline caches. In their Smalltalk implementation, a static type analysis produces a list of inferred receiver types for operation sites. Over multiple program runs, ICTI gleans receiver-type frequency information from ICs and orders the lists of receiver types accordingly. The static analyzer presents the ordered list of receiver types to the developer to help them understand the behaviour of their program more precisely. ICTI mines information from PICs for programmer productivity rather than targeting program efficiency.

Chapter 7

Conclusion

This thesis presents Stub Folding, a novel approach to handling polymorphic inline caches based on fast-path-code equivalence, and a comparative evaluation of Stub Folding and SpiderMonkey's previous inline-caching scheme. The performance results presented in Chapter 5 of this thesis indicate that Stub Folding has merit either as an alternative to the previous inline-caching solution or as a mechanism that can be used in conjunction with the previous approach to handle polymorphic inline caches in dynamically-typed language implementations. This thesis also evaluates two strategies inspired by hardware caching policies. An evaluation of two variations of a Least Recently Used approach suggests that reordering inline cache fast paths based on recency is highly successful in certain cases, but can lead to degradation if not handled correctly. Moreover, an analysis that evicts inline-cache fast paths based on a hit-count frequency does not reliably increase program efficiency for the evaluated benchmark programs. These results demonstrate that there is not any *one-size-fits-all* approach to increase the efficiency of highly polymorphic inline caches.

Additionally, this thesis discusses modern inline caching schemes in the context of JavaScript engines, specifically focusing on CacheIR, an IC design centered around an intermediate representation (IR) that simplifies IC development and enables the reuse of compiled native code through IR matching. This thesis evaluates WarpBuilder, the current JIT compiler front-end used in the SpiderMonkey JavaScript engine that generates specialized code by lowering

CacheIR. The evaluation exemplifies that the combination of CacheIR and WarpBuilder has proven to be a beneficial design that significantly increases the efficiency of executing dynamically-typed programs in SpiderMonkey. This result demonstrates the efficacy of thoughtfully designing dynamically-typed language virtual machines around inline caches.

References

[1] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving javascript performance by deconstructing the type system," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 496–507, ISBN: 9781450327848. DOI: 10.1145/2594291.2594332. [Online]. Available: https://doi.org/10.1145/2594291.2594332. 39, 68

[2] "Areweslimyet." (2023), [Online]. Available: https://firefox-source-docs.mozilla.org/performance/memory/awsy.html (visited on 04/26/2023). 36

[3] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, 2003. DOI: 10.1145/857076.857077. 1

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo, "Runtime feedback in a meta-tracing jit for efficient dynamic languages," in *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS '11, Lancaster, United Kingdom: Association for Computing Machinery, 2011, ISBN: 9781450308946. DOI: 10.1145/2069172.2069181. [Online]. Available: https://doi.org/10.1145/2069172.2069181. 66

[5] S. Brunthaler, "Inline caching meets quickening," in *ECOOP 2010 – Object-Oriented Programming*, T. D'Hondt, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 429–451, ISBN: 978-3-642-14107-2. 1, 22, 67

[6] C. Chambers and D. Ungar, "Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, ser. PLDI '89, Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 146–160, ISBN: 089791306X. DOI: 10.1145/73141.74831. [Online]. Available: https://doi.org/10.1145/73141.74831. 1

[7] C. Chambers, D. Ungar, and E. Lee, "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, ser. OOPSLA '89, New Orleans, Louisiana, USA: Association for Computing Machinery, 1989, pp. 49–70, ISBN: 0897913337. DOI: 10.1145/74877.74884. [Online]. Available: https://doi.org/10.1145/74877.74884. 1, 10, 19

[8] J. Choi, T. Shull, and J. Torrellas, "Reusable inline caching for javascript performance," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 889–901, ISBN: 9781450367127. DOI: 10.1145/3314221.3314587. [Online]. Available: https://doi.org/10.1145/3314221.3314587. 69

[9] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84, Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 297–302, ISBN: 0897911253. DOI: 10.1145/800017.800542. [Online]. Available: https://doi.org/10.1145/800017.800542. 1, 10, 19, 22, 65

[10] T. O. Foundation. "Nodejs." (), (visited on 04/26/2023). 21

[11] A. Gal, B. Eich, M. Shaver, *et al.*, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: Association for Computing Machinery, 2009, pp. 465–478, ISBN: 9781605583921. DOI: 10.1145/1542476.1542528. [Online]. Available: https://doi.org/10.1145/1542476.1542528. 66

[12] N. Glew, S. Triantafyllis, M. Cierniak, M. Eng, B. Lewis, and J. Stichnoth, "Lil: An architecture-neutral language for virtual-machine stubs.," Jan. 2004, pp. 111–125. 66

[13] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983, ISBN: 0201113716. 10

[14] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," *SIGPLAN Not.*, vol. 47, no. 6, pp. 239–250, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254094. [Online]. Available: https://doi.org/10.1145/2345156.2254094. 32

[15] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP '91, Berlin, Heidelberg: Springer-Verlag, 1991. 1, 10, 19, 65, 68

[16] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," *SIGPLAN Not.*, vol. 29, no. 6, pp. 326–336, Jun. 1994, ISSN: 0362-1340. DOI: 10.1145/773473.178478. [Online]. Available: https://doi.org/10.1145/773473.178478. 34, 66

[17] "Javascriptcore javascript engine." (2002), [Online]. Available: https://webkit.org/ (visited on 04/26/2023). 2, 10, 17

[18] "Jetstream2.1." (2022), [Online]. Available: https://browserbench.org/JetStream2.1/ (visited on 04/26/2023). 56

[19] N. Milojković, C. Béra, M. Ghafari, and O. Nierstrasz, "Mining inline cache data to order inferred types in dynamic languages," *Science of Computer Programming*, vol. 161, pp. 105–121, 2018, Advances in Dynamic Languages, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.11.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642317302411. 69

[20] J. D. Mooij. "Cacheir: A new approach to inline caching in firefox." (2017), (visited on 04/26/2023). 17

[21] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN: 1558603204. 25

[22] J. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998. DOI: 10.1109/2.660187. 1

[23] P. G. Salgado. "What's new in python 3.11." (2022), (visited on 04/26/2023). 12, 22

[24] M. Serrano and M. Feeley, "Property caches revisited," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019, Washington, DC, USA: Association for Computing Machinery, 2019, pp. 99–110, ISBN: 9781450362771. DOI: 10.1145/3302516.3307344. [Online]. Available: https://doi.org/10.1145/3302516.3307344. 68

[25] M. Serrano and R. B. Findler, "Dynamic property caches: A step towards faster javascript proxy objects," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 108–118, ISBN: 9781450371209. [Online]. Available: https://doi.org/10.1145/3377555.3377888. 68

[26] M. Shannon. "Pep 659 – specializing adaptive interpreter." (2021), (visited on 04/26/2023). 23

[27] M. Smith, A. Sharma, J. Renner, *et al.*, "Cachet: A domain-specific language for trustworthy just-in-time compilers," in *Workshop on Principles of Secure Compilation*, 2023. 67

[28] "Speculation in JavaScriptCore." (Apr. 2020), [Online]. Available: https://webkit.org/blog/10308/speculation-in-javascriptcore/ (visited on 04/26/2023). 67

[29] "Speedometer2.1." (2022), [Online]. Available: https://browserbench.org/Speedometer2.1/ (visited on 04/26/2023). 56

[30] "Spidermonkey javascript engine." (1996), [Online]. Available: https://spidermonkey.dev/ (visited on 04/26/2023). 2, 10, 13

[31] V. St-Amour and S. Guo, "Optimization coaching for javascript," in *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, J. T. Boyland, Ed., ser. LIPIcs, vol. 37, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 271–295. DOI: 10.4230/LIPIcs.ECOOP.2015.271. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2015.271.

[32] L. Tratt, "Chapter 5 dynamically typed languages," in ser. Advances in Computers, Elsevier, 2009. DOI: https://doi.org/10.1016/S0065-2458(09)01205-4. 1

[33] "V8 javascript engine." (2008), [Online]. Available: https://v8.dev/ (visited on 04/26/2023). 2, 10, 20

[34] A. Wirfs-Brock and B. Eich, "Javascript: The first 20 years," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. DOI: 10.1145/3386327. [Online]. Available: https://doi.org/10.1145/3386327. 10

[35] T. Würthinger, C. Wimmer, A. Wöß, *et al.*, "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013, Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204, ISBN: 9781450324724. DOI: 10.1145/2509578.2509581. [Online]. Available: https://doi.org/10.1145/2509578.2509581. 66