University of California
Santa Barbara

Optimizing JavaScript Engines for Modern-day Workloads

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Madhukar Nagaraja Kedlaya

Committee in charge:

Professor Ben Hardekopf, Chair
Professor Chandra Krintz
Professor Tim Sherwood

December 2015

The Dissertation of Madhukar Nagaraja Kedlaya is approved.

Professor Chandra Krintz

Professor Tim Sherwood

Professor Ben Hardekopf, Committee Chair

November 2015

Optimizing JavaScript Engines for Modern-day Workloads

by

Madhukar Nagaraja Kedlaya

To my parents, my sister, and my wife.

Acknowledgements

I am deeply grateful to my advisor, Prof. Ben Hardekopf for all the support and freedom he has provided throughout my stay as a grad student. His never-ending commitment to encourage his students has helped me through tough times.

I would also like to thank Behnam Robatmili for his constant source of encouragement and thoughtful insights that helped me in my work. I am indebted to Mehrdad Reshadi, Calin Cascaval, and rest of the MuscalietJS team at Qualcomm Research for having faith in me and funding my work for the most part of my PhD.

I am deeply grateful to Prof. Chandra Krintz and Prof. Tim Sherwood for being my constant source of encouragement. I'll always cherish all the conversations that I had with them and the new ideas that sparked during those conversations inspired most of my work described in this dissertation.

I am eternally indebted to my parents, my sister, and my wife for their continuous support and love. I have been lucky to have had a number of extremely brilliant researchers as my labmates. Special thanks to Vineeth Kashyap and Kyle Dewey, with whom I have shared numerous accounts of frustration and a fair amount of moments of joy over the past couple of years.

Curriculum Vitæ

Madhukar Nagaraja Kedlaya

Education

2015	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2015	M.S. in Computer Science, University of California, Santa Barbara.
2008	B.E. in Computer Science and Engineering, Manipal Institute of Technology, Manipal.

Publications

Madhukar N. Kedlaya, Behnam Robatmili, Ben Hardekopf. *Server-Side Type Profiling for Optimizing Client-Side JavaScript Engines.* Dynamic Languages Symposium (DLS), 2015

Madhukar N. Kedlaya, Behnam Robatmili, Calin Cascaval, Ben Hardekopf. *Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines.* Virtual Execution Environments (VEE), 2014 **(Best Paper Award)**

Behnam Robatmili, Calin Cascaval, Mehrdad Reshadi, Madhukar N. Kedlaya, Seth Fowler, Michael Weber, Ben Hardekopf. *MuscalietJS: Rethinking Layered Dynamic Web Runtimes.* Virtual Execution Environments (VEE), 2014

Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, Ben Hardekopf. *Improved Type Specialization for Dynamic Scripting Languages.* Dynamic Languages Symposium (DLS), 2013

Nagy Mostafa, Madhukar N. Kedlaya, Youngjoon Choi, Ben Hardekopf, Chandra Krintz. *The Remote Compilation Framework: A Sweetspot Between Interpretation and Dynamic Compilation.* UCSB Technical Report #2012-03, May 2012

Abstract

Optimizing JavaScript Engines for Modern-day Workloads

by

Madhukar Nagaraja Kedlaya

In modern times, we have seen tremendous increase in popularity and usage of web-based applications. Applications such as presentation software and word processors, which were traditionally considered desktop applications are being ported to the web by compiling them to JavaScript. Since JavaScript is the de facto language of the web, JavaScript engine performance significantly affects the overall web application experience. JavaScript, initially intended solely as a client-side scripting language for web browsers, is now being used to implement server-side web applications (node.js) that traditionally have been written in languages like Java. Web application developers expect "C"-like performance out of their applications. Thus, there is a need to reevaluate the optimization strategies.

Thesis statement: *I propose that by using run-time and ahead-of-time profiling and type specialization techniques it is possible to improve the performance of JavaScript engines to cater to the needs of modern-day workloads.*

In this dissertation, we present an improved synergistic type specialization strategy for optimized JavaScript code execution, implemented on top of a research JavaScript engine called MuscalietJS. Our technique combines type feedback and type inference to reinforce and augment each other in a unique way. We then present a novel deoptimization strategy that enables type specialized code generation on top of typed, stack-based virtual machines like CLR. We also describe a server-side offline profiling technique to collect profile information for web application which helps client JavaScript engines (run-

ning in the browser) avoid deoptimizations and improve performance of the applications. Finally, we describe a technique to improve the performance of server-side JavaScript code by making use of intelligent profile caching and two new type stability heuristics.

Contents

Chapter 1

Introduction

Web has become a convenient medium for distribution of software. Last decade has seen a tremendous rise in popularity of web-applications. With the advent of cloud computing, people have access to massive applications such as search engines and social networks. JavaScript, being the *de facto* programming language of the web, enables the user to interact with such applications. The "write once, run anywhere" slogan once used to describe Java applications is now truly applicable to web-applications written in JavaScript. JavaScript code is portable – it executes on every architecture that is supported by JavaScript engines.

Given the popularity of the language, developers are using JavaScript even outside the context of a browser. In fact, legacy applications are being ported to the web using C/C++/Java/C#-to-JavaScript compilers. JavaScript is being used for server-side programming (node.js) [1], desktop application development(Windows and Linux platforms) [2, 3], game engine scripting [4], and embedded software development [5, 6].

JavaScript is popular among developers and, arguably, has a lower barrier for entry for beginner programmers and allows for rapid prototyping of new ideas. Web-application developers enjoy the dynamism and non-verbose syntax that JavaScript provides. Since

JavaScript is dynamically typed, the programmers do not have to deal with explicitly annotating the source code with types.

The inherent dynamism present in the language, which is endearing to the web community, is also the reason for its slowness. A JavaScript engine has to dynamically check the correctness of the current state of execution while executing the program. This is unlike statically typed languages like Java where the type checker, which is part of the compiler, performs these checks ahead of time before execution. In spite of all these inherent problems, the developers and web application users expect C-like performance from the web applications. A slow engine can give the user an illusion that the whole web site is broken. A responsive web application enabled by a fast engine, is always preferred by users of the application.

A decade ago, JavaScript execution was restricted to interpretation. Since then, JavaScript as a language has matured and has gained in popularity. In order to make JavaScript execution faster, researchers have developed various techniques to improve the performance of JavaScript engine. Newer engines have employed various optimization such as type specialization and have adopted multi-tier adaptive compilation strategies to execute JavaScript faster. Engines have now evolved from simple interpreters to what we call as *runtime systems* with advanced compilers and optimizations. Most engines now implement a compiler that translates the JavaScript code to machine code that executes on the hardware. An adaptive compiler is invoked by the engine whenever a part of the JavaScript source code is deemed *hot*, i.e. there is a high probability that that part of code is going to execute multiple times in the future. Therefore, speeding up the execution of the hot code improves the responsiveness of the application.

Improved performance enables the developers to create new feature rich applications that were not possible before. JavaScript games with advanced graphics, fully-featured office suite, and image processing tools are examples of few applications that have been

made possible due to improvement in JavaScript engine performance. Also, the users benefit from improved response time from the web applications. Server-side JavaScript also benefits from the improved performance. A faster engine can enable better startup time and throughput for the server application.

With the popularity of JavaScript growing by the day, more and more JavaScript code is being added to the web pages pushing the JavaScript engine performance to its limits. Therefore, there is an immediate need for new and innovative dynamic language optimizations and JIT compilation techniques to make JavaScript engines faster.

1.1 Background

JavaScript operates on dynamic values; i.e. types of variables used in the language are statically unknown. JavaScript also allows for object structures to be created dynamically and object properties to be added and removed at runtime. This inherent dynamism present in JavaScript, while useful, presents a significant challenge for efficient language implementation. Because the types of values are statically unknown, the language runtime requires an extra level of indirection: instead of operating directly on values, it operates on special "*dynamic values*" that wrap actual values inside a data structure (boxing) that records the enclosed value's type. To operate on these values the runtime must conditionally branch based on the enclosed value's type, unwrap the enclosed value (unbox) to perform the required operation (which sometimes involves complex type conversion operations), then re-box the result back into a dynamic value. This extra level of indirection not only imposes a large runtime overhead, but also inhibits other optimizations that could take place if the runtime knew the value types ahead of time.

One solution to this problem is *type specialization*. Once pioneered by the Self[7]

language developers, type specialization has become an important optimization that is implemented in most of the JavaScript implementations. Type specialization refers to replacing generic operations that operate on dynamic values in a program with type specific operations. Such types can be classified into two broad categories.

- Primitive types: `integer`, `boolean`, `double`, `undefined`, and `null`
- Non-primitive types: `objects`

It is important to note here that a *type* in the JavaScript realm does not always match the type of the internal value representation of a JavaScript engine. For example, JavaScript exclusively operates on number types which are doubles, whereas the underlying engine can choose to store the value as an integer or double internally. Depending on the underlying architecture of the engine, the JavaScript types can be further broken down into subcategories such as unsigned and signed variants of integers and doubles. The reason for such classification is to allow the optimizing compiler present in the JavaScript engine to generate type specific machine code equivalent to the JavaScript source which runs much faster on the hardware compared to generic operations on dynamic values.

There are two approaches to type specialization– a) type inference and b) type feedback.

Type inference. Type inference enables type specialization without any instrumentation of the code at runtime. The types of some subset of the local variables in a function can be inferred statically before it is compiled and executed. For example, an assignment statement `var a = 0;` means that, according to the language semantics, variable `a` must be of type `integer` immediately after that program point. Any further expressions using `a` can be type specialized based on this deduction as long as `a` is not redefined with a different type. Type inference is usually performed as a whole-program analysis in

statically-typed languages (where type inference was first developed). However, whole-program type inference for a dynamic scripting language is not practical because the type inference is usually done *online* during program execution, and this requires that the type inference process must be extremely fast. Therefore, JavaScript engines usually perform type inference on a per-function basis only for hot functions, detected adaptively during execution. One major drawback of type inference is that the types that are inferred are over-approximations of the actual types observed during run-time. Since the engines perform type inference on a per-function basis, it is not possible for them to effectively infer the types of passed-in arguments, function return types, and global variables. Therefore, the type of such expressions are usually over-approximated to be of any (dynamic) type. This precision loss can significantly effect the quality of the code generated by an optimizing compiler, thereby effecting the performance of the engine.

Type feedback. Type feedback enables type specialization by instrumenting the code at run-time and observing the types actually seen during execution. This process involves instrumenting the runtime to collect and store the types that are observed during execution. We, henceforward, call this process *type profiling* or quite simply *profiling*. For example, an expression `a + b` may imply string concatenation, integer addition, or dictionary update based on the types of `a` and `b`. By profiling the types of `a` and `b` during several executions of this expression, the runtime can type specialize the operation during the subsequent executions for those types that are most often seen. If, for instance, the observed types for `a` and `b` are usually integers then the runtime can insert type specialized code that first checks whether the types of `a` and `b` are `int` and then performs integer addition directly. This type specialization using type feedback comes at a cost. First, collecting type information during the initial execution phases creates overhead with respect to both time and memory. Secondly, types need to be checked

during the course of execution of the program using *guard instructions*. Thirdly, when new types are encountered during execution, the specialized code is no longer valid. The runtime needs to have a recovery mechanism like deoptimization in place in order to handle unexpected types which is usually slow.

Deoptimization is a recovery mechanism where the state and control of execution of a function is transfered from optimized code to unoptimized code. JavaScript engine designers have adopted various deoptimization techniques which were designed to work for engines that were implemented using a low-level programming languages like C or C++ and compile JavaScript to machine code. But these pre-existing deoptimization techniques only work for low-level machine code that is generated by adaptive compilers present in these engines. Language runtimes implemented on top of existing virtual machines, which we call *layered architectures* cannot make use of these deoptimization techniques. These layered architectures are usually implemented on top of a typed stack-based virtual machine– for example, dynamic language runtimes like Rhino, IronJS, IronRuby, JRuby, IronPython, and Jython, which implement JavaScript, Ruby, and Python runtimes respectively, either on top of the Java Virtual Machine (JVM) or the Common Language Runtime (CLR). The inherent typed nature of the underlying runtime imposes specific rules and restrictions on what kind of code can be executed on top of it. These rules make the existing deoptimization techniques impossible to implement on top of such virtual machines, thereby restricting a class of type feedback mechanism that employ deoptimization as their recovery mechanism.

Though having a deoptimization mechanism in place enables type feedback based type specialization, it is a heavy-weight, expensive process that can severely impede the engine's performance. The online profiler does not always capture all the type information required for the compiled code to not deoptimize. So, it is always desirable to avoid deoptimizations by making use of any additional type information collected ahead-of-

time.

In addition to this, it is also important to strike a balance between collecting sufficient profile information and optimizing functions early enough. If too much time is spent collecting profile information, the engine ends up spending precious cycles executing unoptimized code. On the other hand if the function is optimized earlier, it is possible for the online profiler to miss the chance to collect enough profile information, thereby causing the JIT compiler to generate sub-optimal code and, possibly, cause deoptimization. So, it is desirable to have a *type stability* heuristic to determine if the runtime has collected enough profile information to optimize a function and have a guarantee that it will not deoptimize in the future.

1.2 Challenges

We identify the key challenges to implementing type specialization in JavaScript engines and answer the following questions in this dissertation:

1. Can we use type feedback to improve the precision of types inferred?

2. Can we use type inference to reduce the performance overhead associated with type feedback?

3. Can we perform deoptimization-based type specialization in a JavaScript engine implemented on top of a typed, stack-based virtual machine?

4. Can we avoid deoptimization by using profile information that is collected ahead-of-time?

5. Can we use the profile information that is collected ahead-of-time to come up with type stability heuristics?

1.3 Thesis Statement and Dissertation Roadmap

I propose that by using run-time and ahead-of-time profiling and type specialization techniques it is possible to improve the performance of JavaScript engines to cater to the needs of modern-day workloads.

In the rest of the chapters we describe how the run-time and ahead-of-time profiling and type specialization techniques can help improve the performance of JavaScript engines.

1.3.1 Synergistic Type Specialization

First, we address the first two challenges listed in Section1.2. Our solution to the two challenges is a run-time type specialization strategy called *synergistic type specialization*.

In the previous work on improving type specialization by Hackett et al [8], they explore the idea of combining type inference and type feedback for an efficient JavaScript language implementation. However, their approach is limited to combining the type feedback information to help increase the effectiveness of type inference.

We present a *synergistic type specialization strategy* that combines type feedback and type inference in two unique ways to augment and extend each other. First, we use functions' type signatures, i.e. the types of the function arguments at the time of function invocation, as additional inputs to type inference analysis to improve the precision of types inferred. Secondly, we show that type inference can actually be used to support type feedback by using the inferred type information to more intelligently place type profiling hooks, thus significantly reducing profiling overhead and type checks performed during optimized code execution.

We evaluate the synergistic type specialization strategy on a large set of traditional standard benchmarks (including Sunspider, Kraken and V8) and realistic web applica-

tions(including Amazon, BBC, and JS1k demos) which we call web-replay benchmarks. The results show that synergistic type specialization performs better than current state-of-the-art type specialization techniques across all benchmark suites. Moreover, the synergistic type specialization strategy improves the precision of the type inference analysis and greatly reduces the overhead of both type profile sites during profiling and type checks during execution (by about 23.5%). We describe our technique in detail in Chapter 2.

1.3.2 Deoptimization on Top of Typed Stack-based Virtual Machines

Secondly, address challenge 3 from Section 1.2. With this contribution we enable type specialization on top of typed stack-based virtual machines(VMs).

In recent times we have seen several attempts at building an efficient dynamic language implementation [9, 10, 11, 12] on top of typed, stack-based virtual machines like Java Virtual Machine(JVM) and Common Language Runtime(CLR). The obvious advantage of this approach is that the dynamic language implementation can make use of pre-existing features in the underlying virtual machine such as garbage collection, machine code generation, and hardware-specific optimizations for free. The dynamic language designers can, thus, focus on language-specific optimizations such as type specialization without worrying about other aspects like portability and memory management of the runtime system.

In such runtimes, the dynamic language source code is compiled down to the intermediate representation(IR) of the underlying VM. But the typed nature of the underlying VM presents a significant challenge on the kind of type specialization that can be performed on top of it. In particular, pre-existing deoptimization techniques that are designed for language runtimes that compile down to machine code cannot be implemented on top of such VMs.

To solve this problem, we leverage the underlying VM's existing exception mechanism to perform the deoptimization. But using the exception handling mechanism naively does not solve the problem because in stack-based VMs, exceptions throw away the current runtime stack whereas deoptimization should preserve the current state of execution which includes the stack information of the specialized code. This is required in order to re-start execution at the equivalent program point in the unspecialized code. Therefore, we leverage the JavaScript-to-IR generator's bytecode verifier to track and transfer appropriate values on the runtime stack between the specialized code and the unspecialized code when a deoptimization exception is thrown.

We implement our deoptimization technique for *Muscaliet JavaScript* (MCJS), a research JavaScript engine implemented on top of CLR. We compare our technique to an alternate type specialization approach of fast path + slow path recovery mechanism implemented in MCJS and IronJS, an alternate JavaScript implementation on top of CLR. We use standard JavaScript benchmarks such as Sunspider and V8, and several long-running JavaScript applications, and show that our type specialization technique significantly outperforms existing type specialization techniques on layered architecture. As a testament to the effectiveness of our technique, Nashorn [13], the default JavaScript implementation shipped along with Java 8, uses our deoptimization technique for type specialization. We describe our contribution in detail in Chapter 3.

1.3.3 Server-Side Type Profiling

We tackle the fourth challenge presented in Section 1.2 by performing server-side ahead-of-time type profiling of web-applications.

The main idea of server-side profiling is to collect ahead-of-time(AOT) profile information and use it to optimize the execution of JavaScript code in the client JavaScript

engine by reducing deoptimizations and enabling the client JavaScript engine to optimize the potential hot functions aggressively without having to fear increased deoptimizations. The AOT profile is collected by executing the JavaScript application using test inputs in the server. The AOT profile consists of a) types observed by the online profiler during the execution of the program, b) program points where deoptimization occurred, c) function hotness information i.e. the invocation counts for individual functions, and d) deoptimization frequency for hot functions. The AOT profile is then analyzed to weed out unnecessary information that need not be sent to the client. This is important because we need to make sure that the size of the profile information sent to the client is as small as possible. The AOT profile is then annotated into the JavaScript code and sent to the client when requested.

The client JavaScript engine uses the AOT profile information in following ways. First, it aggressively optimizes functions marked as hot in the AOT profile. Secondly, it uses the type annotations added to the source code in addition to the types profiled by the online profiler to optimize the functions. Thirdly, using the deoptimization information present in the AOT profile, the client engine disables certain optimizations that could potentially cause them. Finally, the client engine does not optimize the functions that deoptimize very frequently.

We implement both server-side and client-side modifications in Mozilla's SpiderMonkey JavaScript engine[14]. We evaluate our technique using Octane benchmark suite, JavaScript physics engine libraries, and Membench, a collection of JavaScript-heavy websites and compare against vanilla SpiderMonkey JavaScript engine. We demonstrate considerable improvement in performance of the JavaScript code across all the benchmarks while keeping the AOT profile annotation size to the minimum and showing significant reduction in the number of deoptimizations on the client side.

1.3.4 Accelerating Server-Side JavaScript

Finally, we address the final challenge listed in Section 1.2 in Chapter 4. We solve this challenge in the context of server-side JavaScript execution.

One interesting observation for server-side JavaScript engines is that various instances of server application execute the same code and operate on similar inputs. We make use of this idiom to share the profile information collected by once instance of the server with another to accelerate server-side JavaScript execution. We achieve this by caching the profile information in an external database and sharing it across different instances of the server.

There are two types of information that are used to accelerate the code. The first form of information is called *cachable* information and consists of primitive types, deoptimization information, function hotness information, and information regarding function inlining decisions taken by the optimizing compiler. This is very similar to the AOT profile information described in the previous subsection. In addition to this, we identify other key information that depend on the addresses in the heap such as object shape information which is used to optimize the object property access and object type information which is used during type inference analysis. This information cannot be captured offline in a database and is called *heap-dependent* information. Therefore, we use a notion of *type stability* which describes the point during the execution of a function when it has enough heap-dependent information to be optimized without undue risk of deoptimization. We experiment with two heuristics for determining type stability and explain their relative merits and demerits. Our experiments show that for 7 out of 10 `node.js` benchmarks, the server instances that use the cached profile information show significant improvement in initial throughput compared to the server instances that do not use cached information.

1.4 Permissions and Attributions

1. The content of Chapter 2 is the result of a collaboration with Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf, and has previously appeared in the proceedings of the 9th symposium on Dynamic languages [15]. It is reproduced here with the permission [16] of Association of Computing Machinery (ACM): `http://dl.acm.org/citation.cfm?id=2508177`.

2. The content of Chapter 3 is the result of a collaboration with Behnam Robatmili, Calin Cascaval, and Ben Hardekopf, and has previously appeared in the proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments [17]. It is reproduced here with the permission [16] of Association of Computing Machinery (ACM): `http://dl.acm.org/citation.cfm?id=2576209`.

3. The content of Chapter 4 is the result of a collaboration with Behnam Robatmili and Ben Hardekopf, and has previously appeared in the proceedings of the 11th Symposium on Dynamic Languages [18]. It is reproduced here with the permission [16] of Association of Computing Machinery (ACM): `http://dl.acm.org/citation.cfm?id=2816719`.

Chapter 2

Synergistic Type Specialization

Type feedback and type inference are two common methods used to optimize dynamic languages such as JavaScript. Each of these methods has its own strengths and weaknesses, and we propose that each can benefit from the other if combined in the right way. We explore the interdependency between these two methods and propose two novel ways to combine them in order to significantly increase their aggregate benefit and decrease their aggregate overhead. In our proposed strategy, an initial type inference pass is applied that can reduce type feedback overhead by enabling more intelligent placement of profiling hooks. This initial type inference pass is novel in the literature. After profiling, a final type inference pass uses the type information from profiling to generate efficient code. While this second pass is not novel, we significantly improve its effectiveness in a novel way by feeding the type inference pass information about the function signature, i.e., the types of the function's arguments for a specific function invocation. Our results show significant speedups when using these low-overhead strategies, ranging from 1.2× to 4× over an implementation that does not perform type feedback or type inference based optimizations. Our experiments are carried out across a wide range of traditional benchmarks and realistic web applications. The results also show an average reduction

of 23.5% in the size of the profiled data for these benchmarks.

2.1 Introduction

Researchers and dynamic language implementors have spent considerable effort on creating efficient dynamic language runtimes. The main strategy employed is *type specialization*: replacing the generic code that manipulates dynamic values with code specialized to handle only specific types of values. Of course, this strategy is only effective if the runtime can be guaranteed that the specialized code will only be run on values of the appropriate types. The two differing methods that have historically been used in various language implementations to provide this guarantee are *type feedback* [19] and *type inference* [20].

These two methods have differing strengths and weaknesses. Type feedback uses online type profiling to find code that is (almost) always executed on specific types and specializes the code based on this information. Type profiling provides very precise information which enables many optimizations, but cannot guarantee that the code will never be executed with different types in the future; thus the specialized code must contain type checks that detect and recover when unexpected types are encountered. Type inference, in contrast, deduces value types that must necessarily be correct and specializes the code based on these deductions. While the resulting specialized code does not require any online checks or recovery, the dynamic nature of these languages means that type inference may miss many opportunities for specialization that would be discovered by type feedback.

2.1.1 Key Insights

A natural question to ask is which one of type feedback or type inference is the more effective method. Agesen et al [21] compare these two methods head-to-head in their Self language implementation and found that there was no clear winner. They suggest that future work should explore how to combine these two methods rather than choosing between them. Hackett et al [8] take up this idea to explore combining these two methods for an efficient JavaScript language implementation. However, their combination went in only one direction: they used the type feedback information to help increase the effectiveness of type inference.

Our work shows that there is even more to be gained from combining type feedback and type inference in novel ways. In particular, we present two new strategies for combining the two:

- We show that type feedback can do an even better job of supporting type inference by separating function invocations according to the functions' type signatures, i.e., the types of the function arguments at the time of function invocation.

- We show that, besides using type feedback to aid type inference as has already been explored, type inference can actually be used to support type feedback by using the inferred type information to more intelligently place type profiling hooks, thus significantly reducing profiling overhead.

2.1.2 Contributions

Our specific contributions are:

- We propose a novel language-agnostic way of combining type inference and type feedback for dynamic language runtimes (Section 2.3).

- We improve upon previous schemes for using type feedback to aid type inference by using a function's type signature to distinguish different function invocations (Section 2.4).

- We introduce a new scheme that uses type inference to lower the overhead of type feedback by enabling more intelligent placement of profiling hooks (Section 2.4).

- We implement our proposed schemes in a research JavaScript engine, MCJS. We evaluate this implementation on both the standard performance benchmarks (including Sunspider [22], V8 [23], and Kraken [24]) and on real-world websites (including popular websites like Amazon and BBC) and the JS1k demos [25] (Section 2.5).

We find that this mechanism results in speedups ranging from 1.2× to 4× over an implementation that does not perform type inference and type feedback based optimizations, across standard benchmarks. For web-replay benchmarks, which represent the JavaScript code executed when loading a website, function signature based type inference gives an average speedup of 5%. In the case of the JS1k demo benchmarks, which run for a longer duration, we observe an average speedup of 1.6×. Finally, using the types inferred by type inference, the type feedback in this mechanism inserts 23.5% fewer type feedback sites in the code.

2.2 Related Work

Type inference and type feedback for dynamic scripting languages have been a topic of research for a number of years. In this section, we give a brief overview of the current state of the art approaches in this area.

Compiler developers for the language Self [31] pioneered the concept of using type feedback for optimization of object-oriented dynamic languages. The Self compiler used

an instrumented version of the program being executed to observe the types of the objects or *receiver classes* for every message pass or function call. The program was then specialized for the most frequently observed receiver class. Hölzle et al [32] discuss the implementation of polymorphic inline caches and various strategies used to select the candidate code for specialization. These strategies helped shape the design of the dynamic scripting language runtimes that followed.

PyPy is a mature Python implementation written in a subset of Python called RPython [33, 34, 35]. PyPy contains a tracing JIT compiler that uses runtime profile information to guide its tracing and eventual compilation of code paths. In contrast to this approach our algorithm explores the interdependency of type inference and type feedback to perform type specialization online on a per-function basis. The two approaches are orthogonal to each other and can be combined to improve type specialization.

Rubinius [36] is a Smalltalk-80-style VM and JIT compiler for Ruby. Though not as mature as PyPy, it uses a more traditional method-based JIT compiler. The compiler uses a simple form of type feedback in which they just emit guards to validate type assumptions. They rely on LLVM to perform the bulk of their optimization. Rubinius has a fast compiler that emits bytecode, they then compile the bytecode directly to LLVM IR. By going directly to LLVM IR Rubinius is not able to use Ruby-level semantic reasoning in optimization, thus losing the opportunity to perform high-level optimizations such as type inference.

The Crankshaft compiler [30] in Google's V8 JavaScript engine heavily relies on type feedback to generate specialized code. During the generation of a high-level intermediate representation (Hydrogen), each operation in an expression is specialized based on the observed types. Once this is done a set of other optimizations are performed including a static type inference pass to eliminate unnecessary guards. In contrast to this approach our runtime performs static type inference in two stages, one of which happens before

profiling to reduce the profiling overhead. This not only reduces the amount of unnecessary profile information that is collected but also makes sure that the number of guards is reduced in the generated code. Another distinction is that we supply the function argument types to the type inference pass, which greatly increases its effectiveness.

The Jaegermonkey compiler in Mozilla's Spidermonkey engine performs fast hybrid type inference [8] based on the observed types in the previous runs. The expressions that are not type inferred are encapsulated in a *type barrier* and monitored during runtime. These expressions include global variables, function arguments, object property accesses, array element accesses, and function calls. If the observed type differs from the type used to specialize the code, the whole code is invalidated. Our approach differs from this approach because our algorithm uses the function type signatures as one of the inputs to our type inference algorithm to gain greater precision. Thus, the generated code does not have type barriers or guards around function arguments. In contrast to their approach, our algorithm performs type inference in two stages to significantly reduce the profiling overhead. Another difference between our approaches is that Jaegermonkey's type inference algorithm attempts to infer the type of objects and their fields as well. Our algorithm relies on type feedback for specializing operations on objects and their fields.

2.3 High-Level Overview

In this section, we provide a high-level, language-agnostic description of our proposed ideas. In the next section, we will make the discussion concrete for a specific language (JavaScript) and language implementation (MCJS). We first discuss augmenting type feedback with function signatures to aid the effectiveness of type inference. We then discuss using type inference to aid the performance of type feedback.

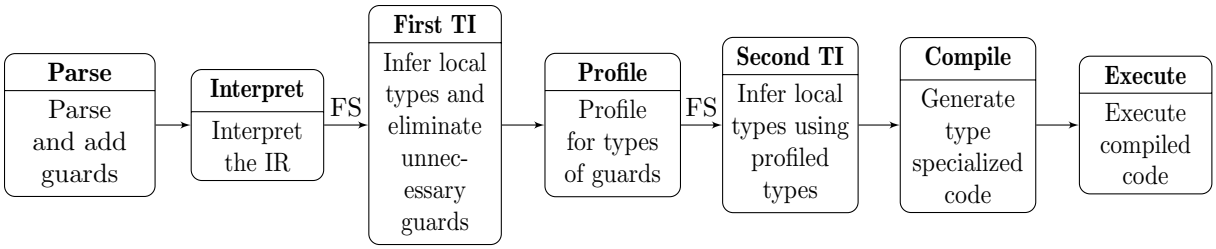

Figure 2.1: A flow graph describing the execution phases of a function. FS stands for function signatures; TI stands for type inference.

Figure 2.1 describes the general workflow of the language runtime system when executing a function, indicating the places where our proposed methods fit in. Our first type inference pass (**First TI**) takes as input the function's signature, i.e., the types of the function arguments for this specific invocation. Different signatures for the same function are handled independently from each other. The phase **First TI** uses the function signature in combination with the standard techniques for type inference. The results are used to place type profiling hooks in the code, and in the phase **Profile** those hooks are used to collect type information that is specific to a given function signature (i.e., the type profiling information is collected and stored separately for each signature of a given function). The phase **Second TI** takes the original function signature along with the collected type profile information and performs a second, more aggressive type inference based on the new information. Finally, the result is used to specialize and optimize the code for further execution.

2.3.1 Function Signatures

Typically, type inference algorithms use the syntactic structure of a function, combined with certain semantic rules of the language, to deduce type information—for example, the result of a left-shift operation is guaranteed to be an integer. However, in a dynamic language there are many operations that do not give any clues about types.

```
func foo(a, b)
{
  var c = a + b;
  var d = global + c + bar()
}
..............
foo(1, 3);
foo("bob", "alice");
foo(2, 5);
```

Figure 2.2: Motivating example

For example, the '+' operator is polymorphic and provides no information to the type inference algorithm. We can improve the available information by providing types for the function's arguments. This idea is inspired by existing schemes for specialized function dispatch based on type signatures, such as multimethods[37, 38]. Our innovation is to make the function signatures an additional input to the type inference algorithm.

Figure 2.2 provides a motivating example. Signature-based dispatch operates as follows: After `foo` becomes hot, during the first call to `foo` a type-specialized version of `foo`'s body is created and then specialized to handle arguments of signature `(int, int)`. During the second call to `foo`, another version of `foo`'s body is created that is specialized to handle arguments of type `(string, string)`. Finally, before the third execution of `foo` the runtime determines that there is a match between the current call's signature and a previously-seen signature. It then re-uses the specialized body for `(int, int)` as the target of the call.

We extend this idea to use the function type signature as an input to the type inference algorithm. Since the function dispatch mechanism ensures that the type signatures are always enforced (i.e., specialized code will never be called with the wrong types), we can rely on these signatures as always being correct and specialize the code for those types

without requiring the type checks or recovery code that is necessary for normal type feedback mechanisms.

2.3.2 Phase First TI: Type Inference → Type Feedback

The goal of type feedback is to provide hints to the runtime and the JIT compiler about the types of variables. To do so, the runtime instruments the function's code with profile hooks that record type information observed during execution. These hooks are placed syntactically during the phase **Parse**, and show up as *guard nodes* in the abstract syntax tree (or IR) anywhere that type information may end up being useful (for example, on either side of a binary operator like '+'). See Figure 2.3 for an example of guard node placement for the function in Figure 2.2.

Type profiling can end up being quite expensive, and so reducing the number of guard nodes to be profiled can significantly improve performance of the profiling phase. Our key observation is that if type inference can already statically determine the type of an expression, then it is unnecessary to profile that expression. The phase **First TI** therefore uses the function signature and standard type inference rules to try and statically infer types for as many of the guard nodes as possible. Any guard node that is successfully typed is marked so that no profiling will be performed on that node during the phase **Profile**.

Of course, there will be many nodes that cannot have their types inferred (or inferring their types is only possible through very complex analysis), such as most object property accesses, untyped array indexing, and function call results. These guard nodes are left unmarked and will be profiled during the profiling phase; the resulting information will feed back into the second type inference pass as described in the next subsection.

For Figure 2.3, suppose that this function is called with a signature `(int, int)`.

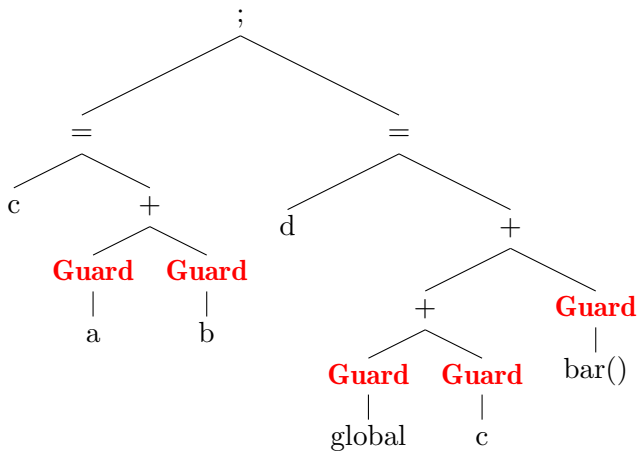

Figure 2.3: Intermediate representation of `foo` before type inference.

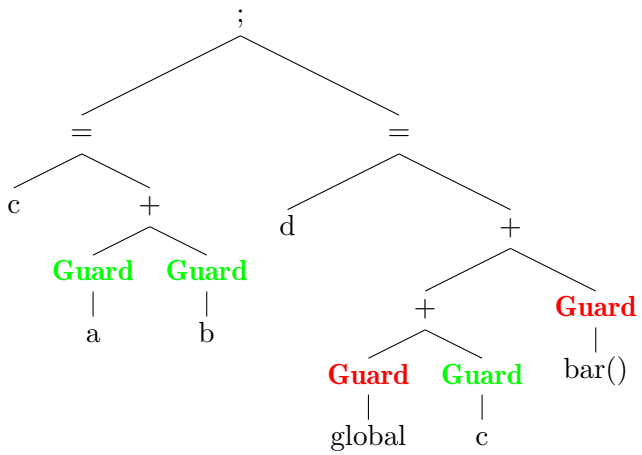

Figure 2.4: Intermediate representation of `foo` after type inference. Statically type inferred guards are in green.

The first type inference pass is then able to infer types for the variables `a`, `b`, and `c`. Therefore, the guards around those variables are no longer useful and do not need to be profiled. Figure 2.4 shows the same function with eliminated guard nodes shown in green.

2.3.3 Phase Second TI: Type Feedback → Type Inference

In the last phase before code generation the runtime uses the type information generated by type feedback to perform a second, more aggressive type inference pass. This pass is identical to the first TI pass except that guard nodes have been annotated with type information supplied by type profiling, and the type inference algorithm uses those annotations instead of attempting to infer the types of the expressions under the guard nodes. This phase is similar to the existing work by Hackett et al [8] except that once again the type information is augmented by the function signature.

In Figure 2.2, suppose that type feedback shows that `global` is always of type `int` and `bar()` always returns a value of type `double`. The second type inference algorithm takes these types into consideration during type inference and thus can infer that variable `d` is type `double`. Since this assumption can be invalidated at any point in the future, the code generator places a type check to enforce the validity of the type feedback information. Consequently, almost all variables and expressions are type inferred at the end and only two guards are placed in this specialization of the function.

2.4 JavaScript Instantiation

In this section we describe a specific instantiation of our proposed ideas for the JavaScript language, using the research JavaScript engine MCJS.

2.4.1 MCJS JavaScript Engine

To evaluate our proposed ideas we use MCJS, a research JavaScript engine written in C♯. This subsection provides a summary of MCJS and its features. MCJS is a layered architecture, as shown in Figure 2.5. This means that the architecture splits

responsibilities across a JavaScript-specific component and a language-agnostic lower-level VM. MCJS specifically uses the .NET Common Language Runtime (CLR) as the lower-level VM, as implemented by Mono [39]. The CLR provides traditional compiler optimizations such as instruction scheduling, register allocation, constant propagation, common subexpression elimination, code generation and machine specific optimizations. In addition, it provides managed language services such as garbage collection.

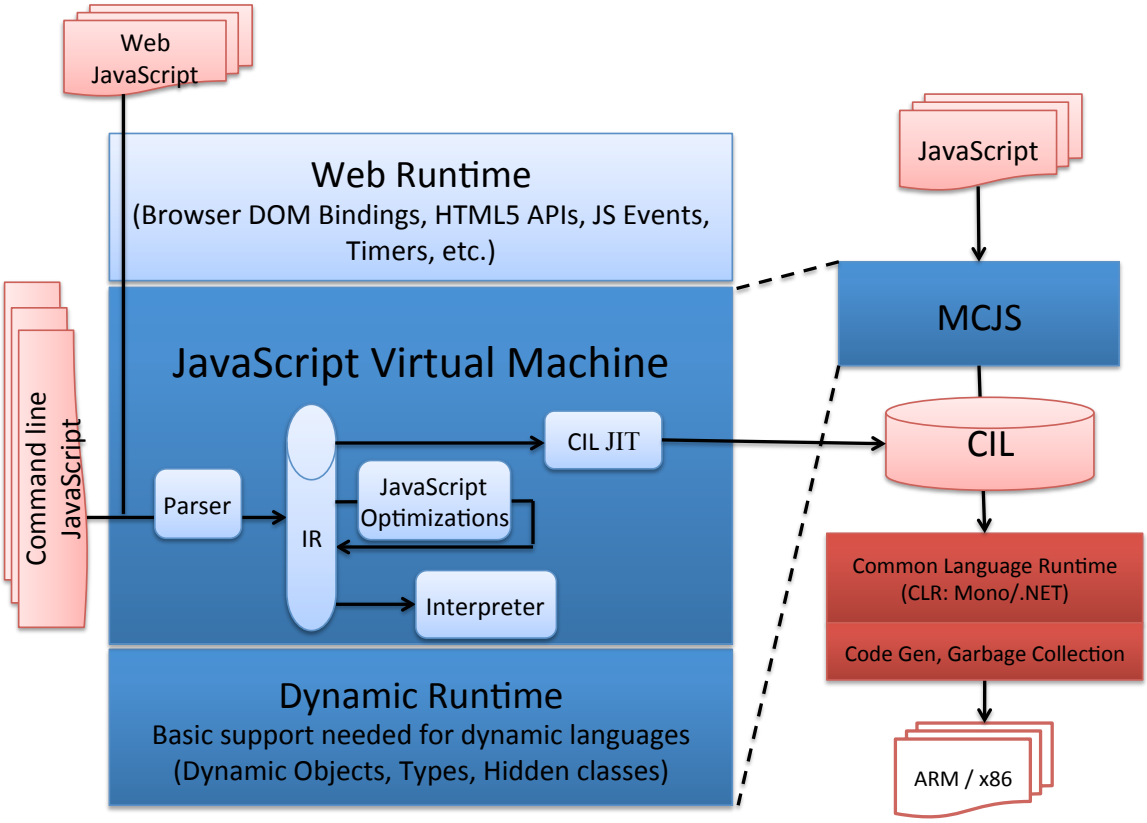

Figure 2.5: MCJS JavaScript Engine Architecture. CIL = Common Intermediate Language, CLR = Common Language Runtime, and IR = MCJS Intermediate Representation

The JavaScript specific layer is a JavaScript VM implemented in C♯. The engine provides the standard dynamic language features such as dynamic values, objects, types, and hidden classes. Additionally, the engine includes the following major JavaScript-

specific components and functionalities:

- **A JavaScript parser** takes in JavaScript code and generates a custom Intermediate Representation (IR) for each function in the code.
- **An interpreter** executes the IR directly for the cold functions during execution.
- **A JavaScript analysis engine** applies JavaScript-specific transformations and optimizations for hot functions. These JavaScript-specific optimizations include type analysis and type inference, array analysis, and signature-based specialization. These transformations augment the IR with extra information for more optimized code generation.
- **A Common Intermediate Language (CIL) bytecode generator** generates optimized CIL bytecodes for hot functions using the IR augmented by the previously mentioned transformations.

For this work, we extend the JavaScript-specific MCJS components to implement the ideas described in the previous section. MCJS already implements signature-based dispatch; the main changes we made were to add the type inference and type profiling phases (i.e., phases **First TI** through **Second TI** as described in the previous section).

2.4.2 Parse Phase: Inserting Guard Nodes

Guard nodes are inserted into a function to indicate where the type profiler should gather type information. Rather than requiring the runtime to transform the code midstream to insert these guard nodes, we have the function parser in the phase **Parse** conservatively inserts guard nodes into the function's IR at every point that may have a dynamic type and may benefit from type feedback. During interpretation these guard nodes are no-ops; their only purpose is to provide a hook for type profiling.

```
function binb2b64(binarray)
{
  var tab = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
            xyz0123456789+/";
  var str = "";
  for(var i = 0; i < binarray.length * 4; i += 3)
  {
    var triplet = (((binarray[i >> 2] >> 8 * (3 -  i %4)) & 0xFF)
                  << 16)
              | (((binarray[i+1 >> 2] >> 8 * (3 - (i+1)%4)) & 0xFF)
                  << 8 )
              |  ((binarray[i+2 >> 2] >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32)
      {
        str += b64pad;
      } else
      {
        str += tab.charAt((triplet >> 6*(3-j)) & 0x3F);
      }
    }
  }
  return str;
}
```

Figure 2.6: `binb2b64` function from the crypto-sha1.js benchmark which is used to convert an array of big-endian words to a base-64 string.The **red** highlighting indicates the presence of Guard nodes around the expressions.

Good candidates for type profiling include binary and unary operations, object property accesses, array element accesses, function calls, and the left-hand sides of assignments. Guard nodes are placed in all of these locations during parse time. However, recall that these are conservative placements—the initial type inference pass, described below, may statically infer types for some of these guarded expressions, in which case the associated guard nodes are marked so that the type profiler will ignore them. As an example, Figure 2.6 shows the function `binb2b64` from crypto-sha1.js. The **red** highlighting indicates the presence of guard nodes around the expressions.

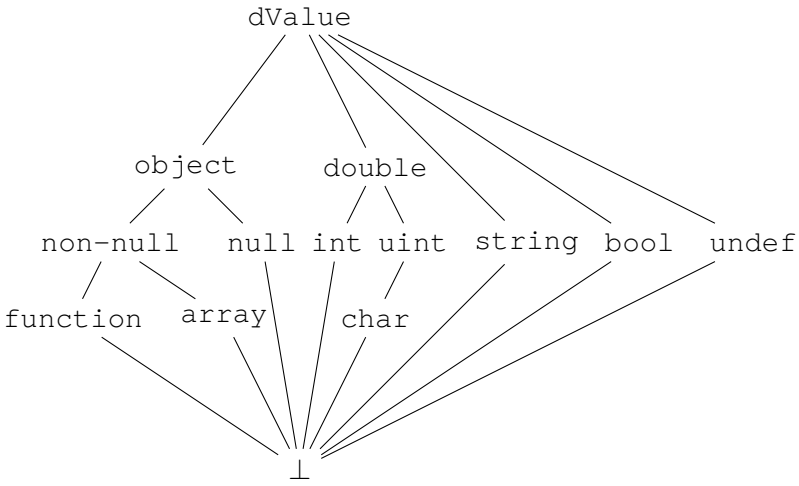

Figure 2.7: Type lattice used by our type inference algorithm.

2.4.3 First TI Phase: Initial Type Inference

Once a function with a particular type signature is deemed hot by the runtime, it is marked as a candidate for further optimization. The first step is an initial type inference pass. This pass will infer as many types as possible using the function signature and the type inference algorithm described by Figures 2.7 and 2.8 and Algorithm 1.

Figure 2.7 shows the type lattice used by the type inference algorithm. The most precise type is $\perp$, indicating an uninitialized value. For objects, we distinguish between function, array, null, and non-null values. For numbers, we distinguish between character, integer, unsigned integer, and double values. The least precise type is `dValue`, which stands for dynamic value—this is the default kind of value to use when the runtime has no static information about the value's type.

Algorithm 1 shows the initialization function for the type inference pass. The local variables are initialized to $\perp$, the parameter symbols are initialized to the types given by the function's type signature, and the global variables are initialized to `dValue` because there is no known information about the possible values of the global variables at this

point in time. The algorithm then places the use sites of these symbols in the worklist.

Algorithm 1 `TypeInference`($\mathcal{S}$, $\mathcal{FS}$), where $\mathcal{S}$ gives the symbols in scope and $\mathcal{FS}$ gives the function's type signature.

```
worklist = []
for s in S do
    switch s.SymbolType do
        case local
            Γ(s) = ⊥
            worklist.add(s.users)
        case parameter
            Γ(s) = τ from lookup(FS, s)
            worklist.add(s.users)
        case global
            Γ(s) = dValue
end for
while worklist.length ≠ 0 do
    e = worklist.pop()
    typeEval(e)                    ▷ Uses the rules from Figure 2.8 to infer types
end while
```

The types of the expressions in the worklist are inferred using a set of typing rules, a selected subset of which are given in Figure 2.8. This subset shows some of the more important inference rules used in the algorithm. The INT and BOOL rules show how constants in the code can be used to type an expression. Rules LSHIFT and GT show how type-specific operations can be used to guide type inference. The ADD rule uses a helper function typeResolve to determine the type of the add operation. The typeResolve function takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation. The VARASSIGN rule generates type constraints. Once all the constraints are collected, they are solved to assign types to the local variables. Finally, the GUARD rules correspond to the guard nodes inserted by the parser. The rules check the type of the expression it encloses; if the expression evaluates to `dValue` then the guard is marked to be profiled by the **Profile** phase, otherwise that

$$n \in \mathit{Num} \qquad b \in \mathit{Bool} \qquad x \in \mathit{Variable} \qquad e \in \mathit{Exp}$$

$$\begin{aligned}\tau \in \mathit{Type} = \{&\texttt{dValue}, \texttt{object}, \texttt{double}, \texttt{non-null}, \texttt{function}, \texttt{array},\\ &\texttt{null}, \texttt{int}, \texttt{uint}, \texttt{char}, \texttt{string}, \texttt{bool}, \texttt{undef}, \bot\}\\ \Gamma \in \mathit{Env} = \mathit{Variable} &\to \mathit{Type}\end{aligned}$$

$$\Gamma \vdash n : \texttt{int} \qquad \text{(INT)}$$

$$\Gamma \vdash b : \texttt{bool} \qquad \text{(BOOL)}$$

$$\Gamma \vdash e_1 \ll e_2 : \texttt{int} \qquad \text{(LSHIFT)}$$

$$\Gamma \vdash e_1 > e_2 : \texttt{bool} \qquad \text{(GT)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e : \tau} \qquad \text{(VARASSIGN)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau = \mathsf{typeResolve}(\tau_1, \tau_2)}{\Gamma \vdash e_1 + e_2 : \tau} \qquad \text{(ADD)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \sqsubset \texttt{dValue}}{\Gamma \vdash \text{guard } e : \tau} \qquad \text{(GUARD 1)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau = \texttt{dValue}}{\Gamma \vdash \text{guard } e : \mathsf{profile}} \qquad \text{(GUARD 2)}$$

Figure 2.8: Selected inference rules used in our type inference algorithm to generate type constraints. Algorithms 1 and 2 conflate the type constraint generation and constraint solving—in the algorithms, the VARASSIGN rule not only generates the type constraint, but also updates Γ with the resultant type and pushes the *users* of the variable x into the worklist. typeResolve is a helper function that takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation.

guard node will be ignored by the **Profile** phase. This rule eliminates many unnecessary guard nodes, significantly increasing the profiler's performance.

As an example, Figure 2.9 shows the function from Figure 2.6 after the initial type inference with the inferred types and the eliminated guard nodes.

2.4.4 Profile Phase

The type profiling phase collects type information at the guard nodes inserted by the parser and marked by the type inference phase 3 as worth profiling. The type information collected by this phase is specific to a particular function *and* function signature. There is only a limited opportunity for profiling the code before it is JITed, therefore we chose to use exhaustive profiling rather than a sampling approach (though this configuration can be modified to use sampling if desired). We employ several heuristics to help minimize the profiling overhead:

- Disable profiling of IR nodes that are highly dynamic in nature: The profiler stops tracking the IR nodes that show highly dynamic nature, such as rapidly changing type information. We observe this behavior in some code snippets which iterate over the fields of an object. For such guard nodes, the profiler records the profiled type as `dValue` and stops profiling them.

- Efficient data structures: While designing the profiler we observed that the performance of the profiler depends heavily on the data structure that is used. In particular, we use an array-based implementation of the profiler which significantly outperforms a dictionary-based implementation.

- Selectively enabling the profiler: We observe that many functions execute only once during the initialization phase of the JavaScript application. Therefore, we enable

```
function binb2b64(binarray⟨array⟩)
{
  var tab⟨string⟩ =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqr
                    stuvwxyz0123456789+/";
  var str⟨dValue⟩ = "";
  for(var i⟨int⟩ = 0; i < binarray.length * 4; i += 3)
  {
    var triplet⟨int⟩ = (((binarray[i >>
    2] >> 8 * (3 - i %4)) & 0xFF) << 16)
                     | (((binarray[i+1 >> 2]
                     >> 8 * (3 - (i+1)%4)) & 0xFF) << 8 )
                     |  ((binarray[i+2 >> 2]
                     >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j⟨int⟩ = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32)
      {
        str += b64pad;
      } else
      {
        str += tab.charAt((triplet >> 6*(3-j)) & 0x3F);
      }
    }
  }
  return str;
}
```

Figure 2.9: `binb2b64` function after the first type inference pass. The **red** highlighting indicates the presence of guard nodes around the expressions that need to be profiled. The **green** nodes indicate that the guard nodes around these expressions are unnecessary and should not be profiled. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm.

the profiler only during the sixth invocation of the function code. By doing this we ensure that we only collect profiles for functions that are potentially hot.

2.4.5 Second TI Phase: Final Type Inference

Once sufficient profile information is collected, the second pass type inference algorithm is performed in **Second TI** phase. In this pass the runtime tries to type the local variables that were not type inferred during the first pass, by using the collected profile information.

Algorithm 2 describes the initialization function of the second pass. This differs from the first pass because we reuse the types inferred by the first pass while initializing the types of the variables in this pass. We check whether the type of a variable is precise enough, i.e., if the type inferred in the first pass is in the set $\mathit{PreciseTypes} = \{$`function`, `array`, `null`, `bool`, `char`, `int`, `undef`, `string`$\}$. If it is, the algorithm initializes the variable to that type, otherwise the algorithm initializes the type of the variable to $\bot$ and adds its users to the worklist. This helps the algorithm converge to a fixpoint faster and avoid inferring types of variables that have already been typed.

The type inference algorithm uses the same inference rules as in Figure 2.8 except for the GUARD rule. The new GUARD rule is:

$$\frac{\tau = \mathcal{P}(\ell)}{\Gamma \vdash \text{guard}^{\ell} e : \tau} \quad \text{(GUARD)}$$

where $\mathcal{P}$ is a function that maps unique labels ℓ associated with guard nodes to the profiled type information. This new rule shows how the profiled type information is used to infer the type of the marked guard nodes. The guards that were not marked (i.e., were not used to gather type information during profiling) are treated as no-ops during this **Second TI** phase.

Algorithm 2 TypeInference($\mathcal{S}$, $\mathcal{FS}$, Γ_1), where $\mathcal{S}$ gives the symbols in scope, $\mathcal{FS}$ gives the function's type signature and Γ_1 is the type environment from initial type inference.

```
worklist = []
for s in S do
    switch s.SymbolType do
        case local
            if Γ1(s) ∈ PreciseTypes then
                Γ(s) = Γ1(s)
            else
                Γ(s) = ⊥
                worklist.add(s.users)
            end if
        case parameter
            if Γ1(s) ∈ PreciseTypes then
                Γ(s) = Γ1(s)
            else
                Γ(s) = τ from lookup(FS, s)
                worklist.add(s.users)
            end if
        case global
            Γ(s) = dValue
end for
while worklist.length ≠ 0 do
    e = worklist.pop()
    typeEval(e)                    ▷ Uses the rules from Figure 2.8 to infer types
end while
```

As an example, in Figure 2.10 we see that `str` is now type inferred to be a string based on the observed types of guards around the `b64pad` and `tab.charAt()` expressions.

2.4.6 Compile Phase: Specialized Code Generation

In this section we discuss the techniques used in generating type specialized Common Intermediate Language (CIL) code in MCJS. After the second type inference pass, the runtime passes the intermediate representation (IR) of the code and the type environment Γ to the specialized code generator. The code generator maps primitive types such as `int`, `bool`, `double`, `char`, and `uint` to native CIL primitives. This ensures that the operations on them can be applied natively and are therefore faster.

After generating code for the expression enclosed in a tagged guard node, a check is added in the code to compare the observed type at execution time with the profiled type. The types inferred in the second pass are valid as long as the checks hold. If the observed type during the execution doesn't match the type for which the code was specialized, the runtime bails out and calls a deoptimization routine. The deoptimization routine captures the current state of the value stack and current values of the variables and reconstructs a new callframe. Once this is done, the execution shifts to the interpreter, which executes the function using the new callframe. This operation is expensive and must be avoided as much as possible. Therefore, capturing accurate profiles is very important.

In the case of the example in Figure 2.10, since `str` is now type inferred as a `string`, the code generator does not add checks around it. With `str` being a local variable, we know its type is only influenced by the observed types of `b64pad` and `tab.charAt()`. Since we already have runtime checks around them, it is unnecessary to check for the type of `str` as well. This small optimization enables the runtime to reduce the number

```
function binb2b64(binarray⟨array⟩)
{
  var tab⟨string⟩ =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
                   xyz0123456789+/";
  var str⟨string⟩ = "";
  for(var i⟨int⟩ = 0; i < binarray.length⟨int⟩ * 4; i += 3)
  {
    var triplet⟨int⟩ = (((binarray[i >> 2]⟨int⟩ >> 8 * (3 - i %4)) &
                                    0xFF) << 16)
                    | (((binarray[i+1 >> 2]⟨int⟩ >> 8 * (3 - (i+1)%4))
                                    & 0xFF) << 8 )
                    |   ((binarray[i+2 >> 2]⟨int⟩ >> 8 * (3 - (i+2)%4))
                                    & 0xFF);
    for(var j⟨int⟩ = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length⟨int⟩ * 32)
      {
        str⟨string⟩ += b64pad⟨string⟩;
      } else
      {
        str⟨string⟩ += tab.charAt((triplet >> 6*(3-j))&0x3F)⟨string⟩;
      }
    }
  }
  return str;
}
```

Figure 2.10: `binb2b64` function after the first type inference pass. The **red** highlighting indicates the presence of Guard nodes which were profiled and ⟨type⟩ indicates the type profiled by the profiler. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm after the first pass. The ⟨type⟩ indicates the type inferred by the type inference algorithm after second pass.

of unnecessary checks in the code. The total number of checks in the final CIL code for `binb2b64` is reduced from nine to seven.

2.5 Evaluation

In this section we describe our evaluation strategy and compare various combinations of our optimizations against a baseline MCJS implementation. Throughout this section, we abbreviate type inference as TI, type feedback as TF, function signatures as FS and guard elimination as GE. We indicate whether the optimizations are enabled or disabled by suffixing + or - respectively. Table 2.1 shows the MCJS configurations on which these experiments are carried out.

We choose the TI- TF+ FS- GE- MCJS configuration because it is in the same vein as V8's strategy for performing type specialization. In this configuration, MCJS performs pure type feedback without any type inference. Though V8's Crankshaft compiler performs various other optimizations and performs a variation of type inference based on the profiled types, we believe this configuration is a fair representation of Crankshaft's type specialization strategy based on the ordering of different phases.

We choose the TI+ TF+ FS- GE- MCJS configuration because, it is in the same vein as the SpiderMonkey's strategy of performing type specialization. In this configuration, MCJS performs type feedback based type inference without considering the types in function signatures. The types of function arguments are initialized to `dValue` during the type inference phase. Though SpiderMonkey's Jaegermonkey compiler performs various other optimizations such as single pass SSA transformation, we believe that MCJS in this configuration is a fair representation of Jaegermonkey's type specialization strategy based on the ordering of different phases.

We emphasize that we *are not* comparing MCJS with Crankshaft or Jaegermonkey

directly. Rather, we compare different type specialization strategies that happen to be used by these engines, among many other optimizations that they implement. No direct conclusions can be drawn from our evaluation about the relative merits of these engines.

2.5.1 Experimental Methodology

We evaluate our optimizations on an AMD FX-6200 Hexa-Core 3.8GHz machine with 10GB RAM. We use Mono 3.0 as our underlying CLR implementation for MCJS. We choose popular JavaScript benchmark suites including Sunspider [22], V8[1] [23] and Kraken [24] as well as JavaScript code from 17 real world websites and web applications to evaluate our implementation. Each of the benchmarks is run 11 times and the data from the last 10 runs is averaged to compute mean performance. We describe the benchmarks in detail in the following subsections.

2.5.2 Standard Benchmarks

Figure 2.11 shows the relative speedup of different MCJS configurations with respect to the base configuration for the Sunspider, V8, and Kraken benchmarks. Since Sunspider benchmarks run for a relatively short period of time, each benchmark is repeated 20 times in a loop. Unlike Sunspider, the V8 and Kraken benchmarks run for a relatively longer period of time. Therefore, they are run without modification.

Sunspider

Figure 2.11 shows that MCJS with the TI+ TF+ FS+ GE+ configuration performs extremely well compared to other configurations for the Sunspider benchmarks, with an average speedup of **4.2**×. The average execution time for the base configuration is 48.4

[1]MCJS does not support typed arrays. Therefore, we do not evaluate our implementation on Octane benchmarks.

MCJS configurations	Description
TI- TF- FS+ GE-	MCJS does not perform type inference or type feedback. We use this as our baseline for measuring speedup.
TI- TF+ FS- GE-	Enabling type feedback and disabling type inference in MCJS.
TI+ TF+ FS- GE-	Enabling type inference and type feedback and disabling function signature based TI in MCJS. In this configuration MCJS performs both the type inference passes with the types from function arguments set to `dValue`. These arguments are profiled during the profile phase.
TI+ TF- FS+ GE-	Enabling type inference and disabling type feedback in MCJS. In this configuration MCJS performs only the first pass type inference.
TI+ TF+ FS+ GE-	Enabling type inference and type feedback in MCJS. In this configuration the guard elimination is not enabled.
TI+ TF+ FS+ GE+	Enabling all of the type-inference-based optimizations in MCJS including guard elimination.

Table 2.1: Table describing various MCJS configurations which we perform our experiments on. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

seconds, and the execution times range from 2 seconds for `bitops-bitwise-and` to 545.3 seconds for `string-tagcloud`.

Type inference provides a significant performance boost for these benchmarks, because they heavily rely on integer arithmetic. The compiler maps those JavaScript numbers that are inferred to be integers to CLR integer primitives. This optimization enables type specialized x86 integer operations in the generated code. This further enables x86 specific optimizations such as common subexpression elimination and faster integer arithmetic using bitwise shift operators. Therefore, benchmarks like `bitops-3bit-bits-in-byte`, `bitops-bits-in-byte` and `math-spectral-norm` perform an order of magnitude better using the TI+TF+ FS+ GE+ configuration.

When compared to the TI- TF+ FS- GE- strategy, all of the type inference based approaches perform better. This can be attributed to the constant boxing and unboxing

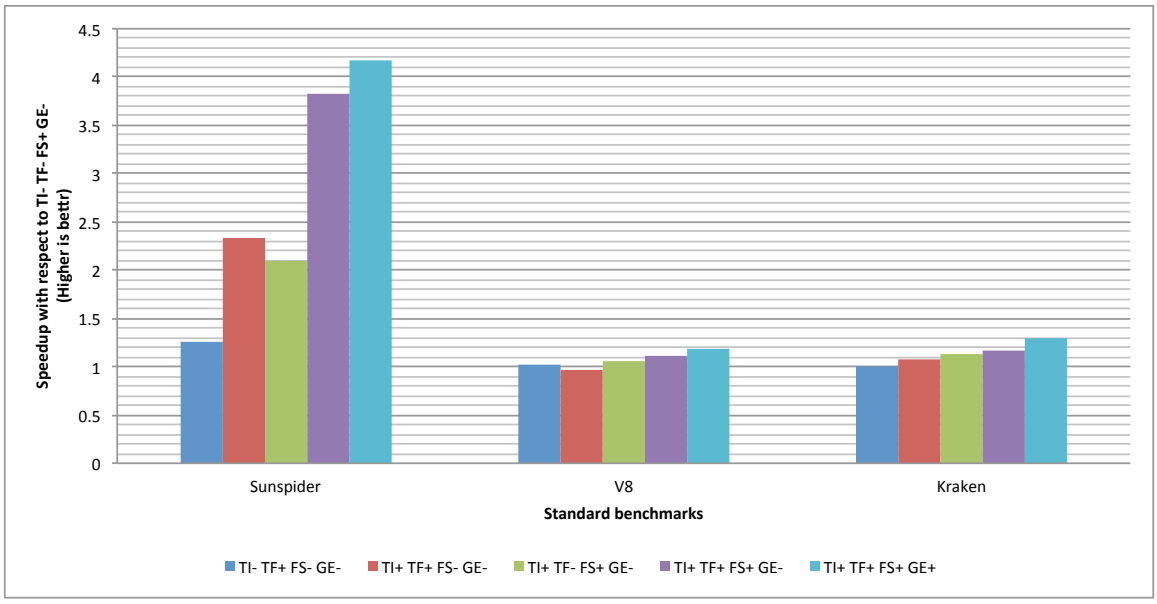

Figure 2.11: Speedup with respect to TI- TF- FS+ GE- configuration for standard benchmark suites: Sunspider, V8 and Kraken. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

of values required by the pure type feedback based approach. When compared to the TI+ TF+ FS- GE- strategy, the configurations with function signatures enabled perform better. This shows that use of function signatures during type inference improves application performance.

Kraken

We see an average speedup of **1.3**× for the TI+ TF+ FS+ GE+ configuration. The run times vary from 0.5 seconds to 559 seconds for the base configuration with an average execution time of 66.1 seconds.

In contrast to Sunspider, the Kraken benchmark suite heavily relies on global arrays and array element manipulation. The TI+ TF+ FS+ GE+ configuration performs well on the crypto subset of the benchmarks, giving an average speedup of **1.6**× over the

base configuration. The optimizations are less effective for the rest of the benchmarks that rely heavily on array element manipulation. Since the type inference algorithm does not infer the types of global symbols, the global array access operations are not very optimized. We are currently investigating ways to extend our type inference algorithm to infer the types of arrays to optimize these benchmarks.

V8

We see an average speedup of around **1.2**× with the TI+ TF+ FS+ GE+ configuration for the V8 benchmarks. The run times vary from 6.1 seconds to 124.1 seconds with an average execution time of 36.1 seconds. The V8 benchmarks pose a different challenge from the Kraken benchmarks since most of them deal with global objects and property accesses. These expressions are also not type inferred by our algorithm. Though MCJS using TI+ TF+ FS+ GE+ performs well for the `splay`, `navier_stokes`, and `raytrace` benchmarks, with an average speedup of **1.5**× over the base configuration, it performs rather poorly on `regexp` and `deltablue`. This poor performance is mostly due to MCJS's inefficient regular expression and string library implementation. Our algorithm does not type the properties of an object and precisely tracking such information is difficult. We are currently working on extending our type inference algorithm to approximately infer types of object properties.

For both the V8 and Kraken benchmarks, MCJS with guard elimination and function signatures enabled does *not* perform significantly better than the other strategies. This is because MCJS engine spends most of the time executing inefficient string and regexp libraries. Therefore, the optimizations due to type specialization do not show any effect on the final execution time.

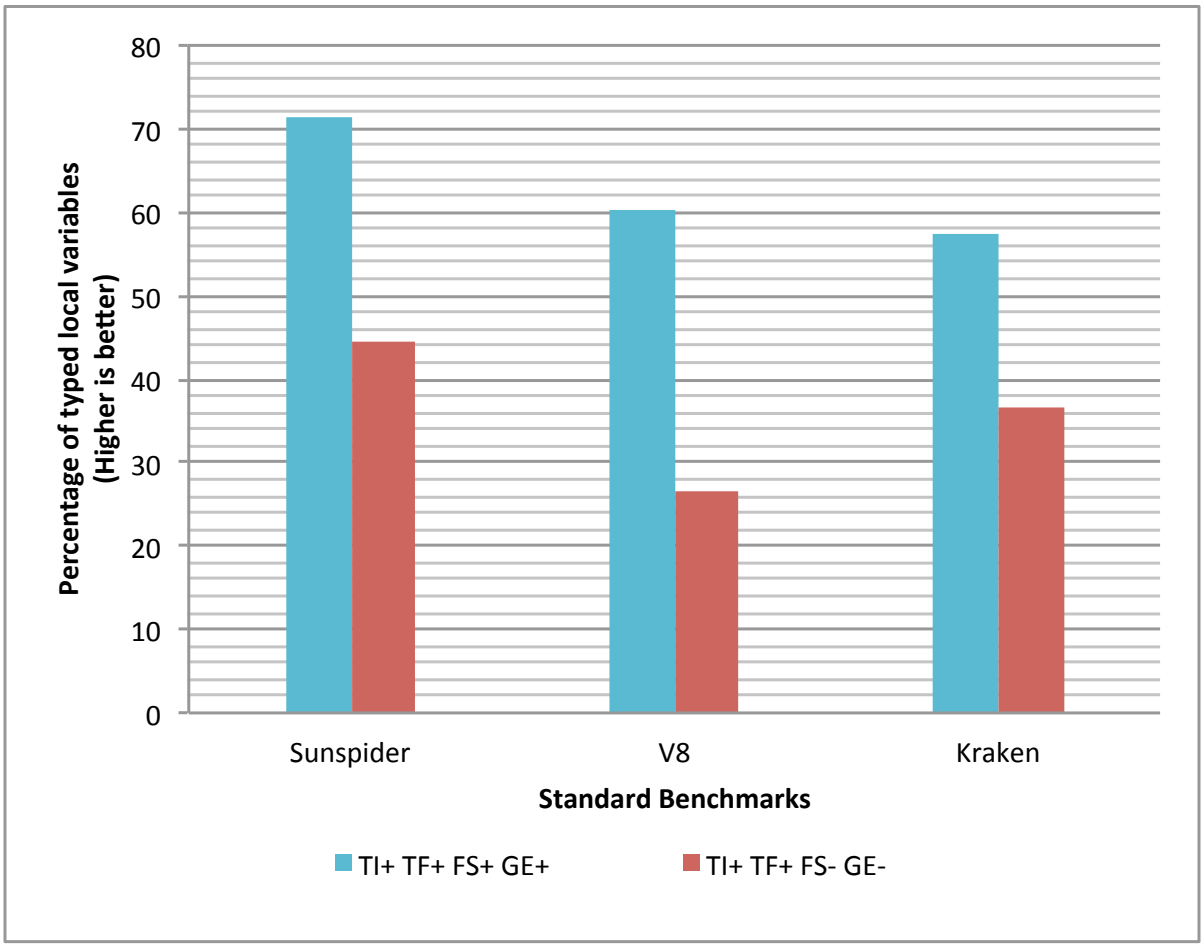

Figure 2.12: Percentage types inferred. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

Effect of Function Signatures

The percentage of type inferred variables is another important metric which shows the effectiveness of including function type signatures in our algorithm. Figure 2.12 shows the percentage of types inferred in the TI+ TF+ FS+ GE+ and the TI+ TF+ FS- GE+ configurations for various benchmark suites. The use of function signatures during type inference improves the percentage of types inferred. For Sunspider, the percentage of local variables that are type inferred increases from **44% to 74%**. There is a significant increase

in this number from **26% to 60%** for the V8 benchmarks. The Kraken benchmarks also show an increase of **21%** in percentage of local variables that are type inferred.

The percentage of types inferred does not directly correspond to the speedup obtained for V8 and Kraken benchmarks, because these benchmarks spend a majority of the time in unoptimized parts of the MCJS engine. For example, the JavaScript standard libraries for strings and regular expressions are extensively used by these benchmarks.

2.5.3 Real-world Benchmarks

Apart from the standard benchmark suite, we test our implementation on 17 real-world websites and web applications. We use the record-and-replay feature of Zoomm [40], a research web browser, to collect the traces of JavaScript that are executed in real-world websites like Amazon, BBC, CNN, Google, Guardian and ESPNCricinfo at load time. These traces are then converted to pure JavaScript files by simulating the DOM objects and their properties in terms of JavaScript objects. Since most of the JavaScript execution happens at page load, the overhead of performing profiler based TI optimizations is not amortized for most of these benchmarks.

Therefore, we also use 11 benchmarks from demos submitted to the JS1k [25] competition. These benchmarks are relatively long running JavaScript applications when compared to the web-replay benchmarks. Though these benchmarks are relatively small in size, we believe that they are representative of core functionalities present in JavaScript heavy web-apps like games and animations. For these benchmarks, the DOM interactions are stubbed out and simulated using pure JavaScript objects. For the benchmarks that require user interaction, the events are simulated by providing them a fixed set of JavaScript event objects in a loop. The `setTimeout` and `setInterval` functions are replaced by loops that call the supplied function for a fixed number of iterations. We

Benchmarks	Description
Kaboom	JavaScript version of the classic arcade game Boom
Mandelbrot	Animation of classic mandelbrot with user clickable interface for zooming.
Spring pond	Algorithm that simulates the evolution of species of fishes in a pond and survival of the fittest.
Tetris	JavaScript version of the classic Tetris game.
Wave graph	Graph plotting application that plots continuous multicolored sinusoidal waves.
Breakout	JavaScript version of the paddle and ball game.
Conways	Animation simulating the Conway's game of life algorithm.
Flying windows	Animation showing flying windows.
Loading spinner	Spinner animation shown during page load.
Sierpinski gasket	3D representation of Sierpinski gasket fractal.
Analog clock	Analog clock written in pure JavaScript and HTML.

Table 2.2: Table describing the nature of the JS1k demos used as benchmarks.

describe the nature of these benchmarks in Table 2.2.

Web-replay benchmarks

Figure 2.13 shows that our profiler based optimizations do not speedup the web-replay benchmarks, which are the first six benchmarks in the graph (indeed we see slowdown in some cases). This property is seen across all configurations which use type feedback, i.e., TF+. This is mainly because the specialized code is not executed long enough to amortize the overhead caused by profiling. Though web-replay benchmarks execute for an average of 4.1 seconds, most of the functions are executed only a few number of times. Therefore, the web-replay benchmarks are not optimized by type feedback.

But MCJS with function signature based type inference, i.e., TI+ TF- FS+ GE- (green bar) configuration shows speedup in most of the benchmarks with an average speedup of **5%**. This shows that a quick function signature based type inference performs well even during page load.

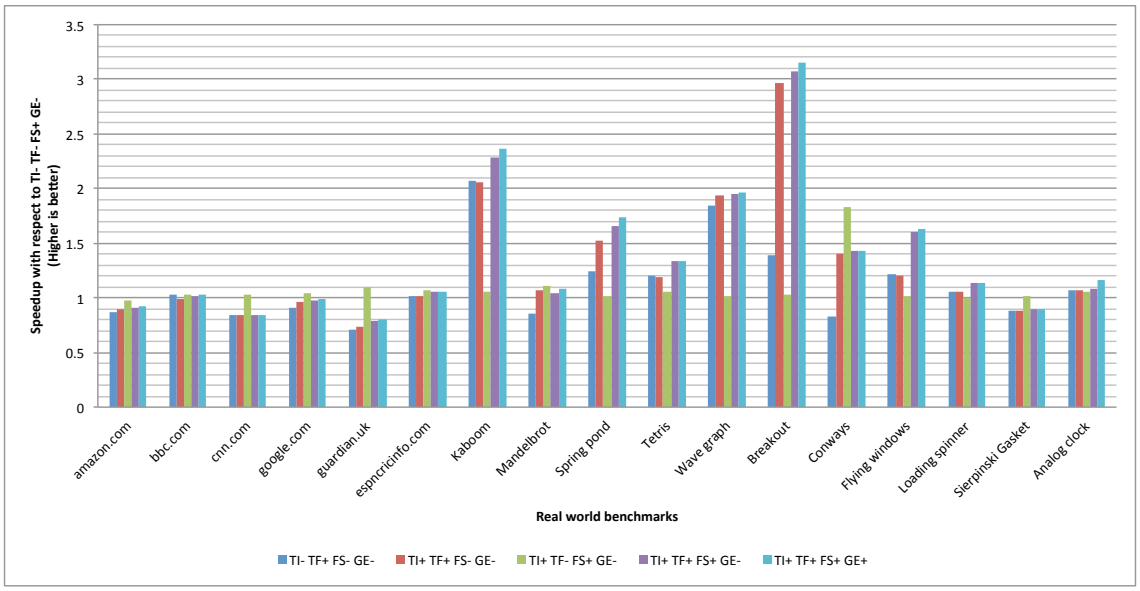

Figure 2.13: Speedup of various configurations of MCJS with respect to TI- TF- FS+ GE- for real world benchmarks. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

JS1k Demos

The final 11 benchmarks shown in Figure 2.13 are the JS1k demos. These benchmarks run for a relatively longer period of time with an average run-time of 11.3 seconds. Most of these benchmarks perform well with type feedback enabled, except for a few exceptions like `Conways` and `Sierpinski gasket` where type inference without type feedback performs better.

Like most of the JavaScript in popular websites, these benchmarks are minified using JavaScript minifiers like Google Closure Compiler [41] or JSCrush [42]. In addition to minifying, some of the benchmarks use global symbols in order to save space. The rest of them maintain local symbols in the functions. Therefore, we see varied behavior across configurations. `Kaboom`, `Spring pond`, `Tetris`, `Wave graph`, `Breakout`, and `Flying windows` use global variables heavily in their code. Therefore, type infer-

ence with type feedback performs better than type inference without type feedback.

In comparison to other strategies, our best strategy with all features enabled (TI+TF+FS+GE+) consistently performs better, especially on `Kaboom`, `Spring Pond`, `Breakout`, and `Flying Windows`. For `Breakout` and `Spring Pond`, type inference benefits both the TI+TF+FS-GE- strategy and our best strategy as compared to TI-TF+FS-GE-. However, for all these specific benchmarks, the main advantage of our strategy compared to other strategies stems from the combination of signature based type inference and guard elimination as can be seen on Figure 2.13.

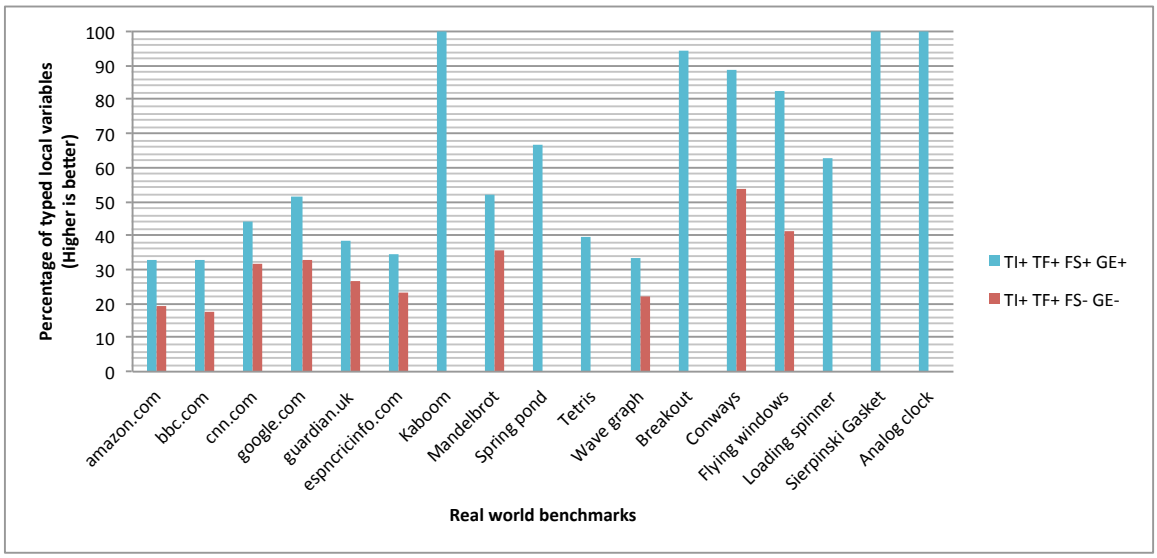

Figure 2.14: This graph shows percentage of typed local variables for configurations TI+ TF+ FS+ GE+ and TI+ TF+ FS- GE+ for real-world benchmarks. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

Effect of Function Signatures

Figure 2.14 shows that the effect of using function signatures in type inference is similar to that observed for the standard benchmark suite. For the benchmarks that heavily rely on global variables, the use of function signatures during the type inference does not

always give major benefit. For example, for benchmarks `Kaboom`, `Mandelbrot`, `Wave graph`, and `Conways`, even though the percentage of types inferred with function signatures is higher than without, we do not see much difference in execution time. This can be attributed to typing of variables that are non-critical for performance. In these benchmarks we observe that the the performance sensitive parts of the code like loops use a combination of global variables and local variables that are not typed in the first type inference phase.

2.5.4 Effect of Guard Elimination

The number of guards eliminated is an indicator of the effectiveness of the profiler. We measure this by collecting the number of unique guard nodes that are profiled during the execution of the program. We then compare the number of guard nodes eliminated due to the guard elimination technique.

Table 2.3 shows the percentage of guards eliminated due to guard elimination for each of the benchmark suites. On an average guard elimination results in **23.5%** reduction in guards profiled during the profiling phase. As Figure 2.11 and Figure 2.13 show, elimination of guards improves performance. It also helps reduce the amount of the information collected during runtime, thereby reducing the total memory used by the application. The elimination of guards (hence conditional control paths) from the CIL also creates more optimization opportunities for the underlying VM's code generator.

2.5.5 Effect of First TI Phase

There are two advantages of the first TI phase. First, it helps in reducing the number of guards that are profiled. This reduces the time spent during collection of type profiles as well as the time required to do the dynamic type checks. Secondly, it speeds up the

Benchmarks	GE-	GE+	% reduction in guards
Sunspider	1203	872	27.5
V8	2053	2004	2.4
Kraken	294	177	39.8
Web replay	62	50	19.4
JS1k Demo	177	127	28.2
Average			**23.5**

Table 2.3: Percentage of guards reduced due to guard elimination for each kind of benchmarks. GE+ indicates the configuration TI+ TF+ FS+ GE+ and GE- indicates the configuration TI+ TF+ FS+ GE- shown in Table 2.1. Numbers on columns two and three are absolute numbers of guards (type checks) across the corresponding benchmark suite and the configuration.

operations involving the variables that are type inferred, during the profiling phase. Our best strategy (TI+ TF+ FS+ GE+) is 2.1× and 1.1× faster than the strategy without the first TI phase for Sunspider benchmarks and real-world benchmarks respectively.

2.6 Conclusion

We explore the phase interdependency between the two most important methods used for type specialization of dynamic languages, type feedback and type inference. Our analysis shows that type feedback can improve the accuracy of type inference (as shown in previous work), but also that type inference can also significantly reduce the overhead of type feedback (during profiling) and type checks (during execution), resulting in overall more accurate and faster type analysis.

This chapter proposes a novel strategy for combining type inference and type feedback in a way that reduces the overhead and improves the performance of both methods. In this strategy, two passes of type inference are applied, both before and after type feedback (profiling). The first type inference pass significantly reduces the profiling overhead during the type feedback phase. On the other hand, the reduced type feedback collected

is then used by the second type inference pass to highly specialize the generated code. The key enabler for this multi-phase efficiency is syntactic guard instructions inserted at parse time into the IR, representing the possible profiling sites. These guards nodes are pruned and marked during the first type inference and type profiling phases. This combined strategy also employs function type signatures to further improve the accuracy and reduce the overhead of both type inference and type feedback methods.

We evaluate the proposed combined function signature based type inference and type feedback strategy on a large set of traditional benchmarks (including Sunspider, Kraken and V8) and realistic web application benchmarks (including Amazon, BBC and JS1k demos). The results show that our proposed method speeds up the standard benchmarks by between 1.2× and 4.2× over the base implementation that does not perform type feedback or type inference based optimizations. For web-replay benchmarks, which represent the JavaScript code executed during website load, simple function signature based type inference gives an average speedup of 5%. In the case of JS1k demo benchmarks, which run for a longer duration, we observe an average speedup of 1.6×. Further more, this combined strategy is able to infer the types of symbols in the hot functions very accurately (between 60% and 80% of all variables) for both standard benchmarks and web applications. Moreover the combined strategy greatly reduces the overhead of both type profile sites during profiling and type checks during execution (by about 23.5%).

Chapter 3

Deoptimization on Top of Typed, Stack-based Virtual Machines

We are interested in implementing dynamic language runtimes on top of language-level virtual machines. Type specialization is a critical optimization for dynamic language runtimes: generic code that handles any type of data is replaced with specialized code for particular types observed during execution. However, types can change, and the runtime must recover whenever unexpected types are encountered. The state-of-the-art recovery mechanism is called *deoptimization*. Deoptimization is a well-known technique for dynamic language runtimes implemented in low-level languages like C. However, no dynamic language runtime implemented on top of a virtual machine such as the Common Language Runtime (CLR) or the Java Virtual Machine (JVM) uses deoptimization, because the implementation thereof used in low-level languages is not possible.

In this chapter we propose a novel technique that enables deoptimization for dynamic language runtimes implemented on top of typed, stack-based virtual machines. Our technique does not require any changes to the underlying virtual machine. We implement our proposed technique in a JavaScript language implementation, MCJS, running on top

of the Mono runtime (CLR). We evaluate our implementation against the current state-of-the-art recovery mechanism for virtual machine-based runtimes, as implemented both in MCJS and in IronJS. We show that deoptimization provides significant performance benefits, even for runtimes running on top of a virtual machine.

3.1 Introduction

Language-level virtual machines (VMs) provide a number of advantages for application development. These advantages extend to implementing language runtimes on top of existing VMs, which we call *layered architectures*—for example, dynamic language runtimes like Rhino, IronJS, IronRuby, JRuby, IronPython, and Jython, which implement JavaScript, Ruby, and Python runtimes respectively, either on top of the Java Virtual Machine (JVM) or the Common Language Runtime (CLR).

However, VMs can also impose performance penalties that make language implementation unattractive. These penalties include not only VM overheads, but also *opportunity costs* arising from optimizations common to native runtime implementations[1] but difficult or impossible within a VM. Our goal in this work is to alleviate an important opportunity cost for implementing dynamic language runtimes on top of VMs. Specifically, we introduce a novel technique for *deoptimization* on typed, stack-based VMs that enables efficient type specialization, a critical optimization for dynamic language runtimes (explained further in Section 3.2).

Why Implement Languages on a VM? There are many advantages to using a layered architecture. Layered architectures provide nice program abstractions, free optimizations, and highly-tuned garbage collection, which are all required for a performant engine.

[1]By which we mean runtimes implemented in a low-level language such as C and compiled to native binaries.

Leveraging an existing VM allows the language developers to focus on language-specific optimizations without bothering with machine-specific optimizations that are handled by the existing VM. Layered architectures also offer a good platform for experimenting with new language features and different optimization techniques for language runtimes. Finally, using a layered architecture enables interoperability between different languages implemented on the same runtime.

Opportunity Costs. The existing VM often imposes restrictions on the language developer that can prevent important optimizations. For example, a key optimization for dynamic languages is *type specialization*, which uses dynamic profiling to specialize code based on observed type information. Type specialization is unsound and thus requires a recovery mechanism to deal with unexpected types by transferring execution from the type-specialized code to the original unspecialized code. However, the very nature of typed, stack-based VMs such as the JVM or CLR means that the most effective known recovery mechanism, deoptimization, cannot be implemented using any known techniques that are used in native runtimes [43, 44, 45, 46].

Key Insights. We have developed a novel technique for effective deoptimization on typed, stack-based VMs. Our key insight is that we can leverage the VM's existing exception mechanism to perform the deoptimization. Doing so is non-trivial, because exceptions throw away the current runtime stack whereas deoptimization should preserve the stack information from the specialized code in order to re-start execution at the equivalent program point in the unspecialized code. Our technique leverages the code generator's bytecode verifier to track and transfer appropriate values on the runtime stack between the specialized code and the unspecialized code when a deoptimization exception is thrown.

Contributions. Our specific contributions are:

- We describe a novel deoptimization technique to enable type specialization for dynamic language runtimes running on top of a typed, stack-based virtual machine. (Section 3.3)
- We describe a specific instantiation of this technique for MCJS[2] [47], a JavaScript engine implemented on top of the CLR. (Section 3.4)
- We evaluate our MCJS implementation and compare against (1) a non-type specializing version of MCJS, (2) a type specializing version of MCJS using an alternate fast-path + slow-path recovery technique, and (3) IronJS [10], a JavaScript engine implemented using the DLR (which performs type specialization using the fast path + slow path technique). We use both standard benchmarks (i.e., Sunspider and V8) and long-running web JavaScript applications, and show that our deoptimization technique significantly outperforms existing type specialization techniques for layered architectures. On an average (geomean) our deoptimization technique is 1.16× and 1.88× faster than MCJS with fast-path + slow-path recovery technique and IronJS respectively. (Section 3.5)

Before describing our technique, we provide background on type specialization, the two dominant recovery mechanisms used by type specialization, and the challenges they face when implemented on top of a VM (Section 3.2).

3.2 Type Specialization

In this section we give background on type specialization, the two dominant recovery mechanisms (*fast path + slow path* and *deoptimization*) used to implement type spe-

[2]http://www.github.com/mcjs/mcjs.git

cialization, and the challenges faced by these techniques when implemented on top of VMs.

3.2.1 Type Specialization

Dynamic languages are dynamically typed, i.e., a variable can refer to values of different types at different points during a program's execution. However, dynamic language runtimes implemented in a typed language must declare a single type for each variable in the underlying implementation. Therefore, runtimes must wrap base values (e.g., integers, booleans, strings, etc) inside a wrapper type called a *DValue*, which stands for "dynamic value". Wrapping a base value inside a DValue is called *boxing*, and extracting a base value from a DValue is called *unboxing*.

The semantics of dynamic language operations depend heavily on the types involved. For example, the simple expression `a + b` can mean many different things depending on the types of `a` and `b` at the time the expression is evaluated. The runtime must unbox `a` and `b` to determine the types of the wrapped base values, perform the appropriate operation, and then box the result back into a DValue. Thus every expression encountered during execution requires unboxing values, performing a series of branch conditions based on type, performing the desired operation, and finally boxing a value. These operations tend to dominate the execution time of any dynamic language program.

In response, dynamic language implementors have developed an optimization called *type specialization*. During execution the observed types of each variable's values are monitored. The runtime then dynamically generates code that is specialized for the observed types. In the previous example, if `a` and `b` are always observed to hold integer values, then the runtime can generate specialized code that declares them to be `int` types instead of DValues and thus avoid all of the unboxing, branching, and boxing.

```
if (GetType(a) == Int && GetType(b) == Int) {
  c = ToDValue(IntAdd(a.ToInt(), b.ToInt()));
}
else { // Slow path
  c = GenericAdd(a, b);
}
// c is of the type DValue here.
```

Figure 3.1: C-like pseudocode representing the fast path + slow path approach for the statement `c = a + b` where `a` and `b` are observed to be integers.

```
if (GetType(a) == Int && GetType(b) == Int) {
  c = IntAdd(a.ToInt(), b.ToInt());
}
else {
  // Jump to deoptimization routine.
}
// c is of the type int here.
```

Figure 3.2: C-like pseudocode representing the deoptimization approach for the statement `c = a + b` where `a` and `b` are observed to be integers.

However, this optimization is unsound—for example, while `a` and `b` have been integers so far, they may hold strings at some point later in the execution. Runtimes that use type specialization must have some sort of *recovery mechanism* that detects unexpected types and falls back to the standard, generic evaluation algorithm. There are two dominant approaches for this recovery mechanism; we describe each below along with their challenges with respect to being implemented on VMs.

3.2.2 Recovery Option 1: Fast Path + Slow Path

For the *fast path + slow path* recovery mechanism, type-specialized code is guarded by a conditional that tests the current types of the specialized variables. If the current types match the expected types then the true branch containing the type-specialized code

is taken, otherwise the false branch containing the generic, unspecialized code is taken. In pseudocode, where `variable` is a DValue:

```
if (unbox(variable).type == type T) {
  T variable' = unbox(variable)
  // fast path: specialized code for type T
  // computes the result using variable'
  box(result)
}
else {
  // slow path: unspecialized code computes
  // the result using variable
}
// use result
```

Notice that variables are still unboxed and boxed for the fast path; this is because the type of `result` must be the same regardless of whether the fast path or slow path is taken. However, there may be multiple operations contained in the fast path and so the cost of boxing and unboxing is amortized; in addition, there is no branching on types in the fast path.

Figure 3.1 gives C-like pseudocode showing how the runtime implements the fast path + slow path operation for a simple binary add operation. Based on the previously observed types of `a` and `b`, say `int` and `int`, the runtime generates code to perform integer addition in the fast path and a generic add operation in the slow path.

Challenges. This technique is the one used in current layered architectures for dynamic languages that perform type specialization, such as IronJS and MCJS. There is no technical difficulty in implementing it, however the constant boxing and unboxing severely limits the benefits of type specialization. Deoptimization is known to out-perform fast path + slow path in native code implementation of dynamic language runtimes; however as we describe below deoptimization is difficult for VMs.

3.2.3 Recovery Option 2: Deoptimization

For the *deoptimization* recovery mechanism, type-specialized code is again guarded by a conditional that tests the current types of the specialized variables. The key difference is that the fast path and slow path are not contained inside the branches of the condition; instead, the slow path is placed in an entirely separate routine. If the condition fails then control leaves the current, type-specialized routine and jumps to the generic, unspecialized routine, where it resumes execution at the unspecialized program point that is equivalent to the specialized program point where the type mismatch was detected. In pseudocode, where `variable` is a DValue:

```
if (unbox(variable).type == expected type T) {
  T variable′ = unbox(variable)
  // fast path: specialized code for type T
  // computes the result using variable′
}
else {
  // jump to equivalent program point in
  // unspecialized code
}
// use result
```

The benefit of this approach is that the remaining code in the routine can assume that the fast path succeeds, and hence we do not need to box the result—we can leave it as whatever type it was specialized to, because if it wasn't supposed to be that type then the code would have jumped completely out of the specialized routine and into the unspecialized routine.

Figure 3.2 gives C-like pseudocode describing the deoptimization approach for the statement `c = a + b`. Similar to the fast path + slow path approach, the guard condition checks whether the observed types of `a` and `b` are integers. If so, the runtime unboxes the integer values of `a` and `b` and performs the integer addition operation. This constitutes the fast path. A difference here with respect to the fast path + slow path

approach is that resultant value is not boxed back into a DValue before assigning it to `c`. Instead, the type of `c` is initialized to be an integer. This prevents further unboxing of `c` when it is used later in the function. The deoptimization code captures the current state of execution of the code and transfers it to either an interpreter or to non-optimized compiled code.

Challenges. Deoptimization has been used in native code implementations of dynamic language runtimes. However, the techniques used there do not translate to typed, stack-based VMs such as the CLR or JVM. Native code uses either *code patching/on-stack replacement* or *long jumps*. In the former strategy, deoptimization is implemented by dynamically replacing the specialized code in the runtime stack with the generic unspecialized code. However, in managed VMs runtime modification of generated functions is not allowed. In the latter strategy, deoptimization is implemented as a long jump to the unspecialized code. However, in managed VMs long jumps are not allowed, for two reasons: first, it disables all optimizations that can be performed within a basic block, and second, these jumps can violate the *Gosling principle* which dictates that stack-based VMs should guarantee the *typestate* at any given program point. Typestate refers to the types of a function's local variables and the types of the values in the operand stack; stack-based VMs enforce the Gosling principle to help ensure correctness and performance. Thus, implementing the deoptimization strategy for type specialization using known techniques is not possible without modifying the underlying VM.

3.3 Deoptimization on Layered Architectures

In this section we give a high-level overview of our approach to solving the deoptimization problem on layered architectures. We discuss two aspects: (1) how to jump

from the specialized code to the correct place in the unspecialized code; and (2) how to transfer the current state from the specialized code to the unspecialized code.

Jump to Unspecialized Code. When specialized code detects a type mismatch, it must jump from the current program point in the specialized code to the equivalent program point in the unspecialized code. As explained in Section 3.2, we cannot use the standard techniques of code patching or long jump to implement this behavior. Instead, we leverage the underlying VM's exception-handling mechanism. The jump from specialized code is done by throwing a `GuardFailure` exception. The body of every optimized method is wrapped in a try block, and deoptimization for every expression in that body is handled in a common catch block. Figure 3.5 illustrates the structure of the specialized code that is generated for a specific example.

The catch block must then transfer control to the unspecialized code, specifically the point equivalent to where the exception was thrown in the specialized version. To achieve this, we assume that the dynamic language runtime implements something like a subroutine-threaded interpreter [48]. A subroutine-threaded interpreter implements each operation of the program (e.g., reading a value of a variable, or performing binary addition) as a separate, unspecialized subroutine implemented in the underlying VM bytecode; each subroutine returns a pointer to the next subroutine that should be executed, and so interpretation consists of a series of subroutine calls with each call returning the address of the next subroutine to call.

Assuming the interpreter is subroutine-threaded, each language expression has an unspecialized implementation in the form of a subroutine with a known address. At each deoptimization guard, a pointer to the appropriate expression's subroutine is hardcoded into the thrown exception's value. The catch block then calls the appropriate subroutine to transfer control to the unspecialized code. We illustrate this process with an example

in Section 3.4.

State Transfer. It is not sufficient to simply transfer control from the specialized code to the unspecialized code; we must also transfer the current state of the program, i.e., the values of the local variables on the runtime stack *and* the values on the operand stack used to store intermediate values during expression evaluation. Transferring the local variables is straightforward: we insert code immediately before the `GuardFailure` exception to read the values of each local variable and store them in a separate data structure shared by both specialized and unspecialized code. We describe such a data structure in Section 3.4.

The tricky part of state transfer is the operand stack. This stack is cleared whenever an exception is thrown, and its values are not stored in local or temporary variables. For example, suppose while evaluating the expression `a + b + c` that there is a deoptimization guard around `c` that throws an exception. The value of `a + b` resides (only) in the operand stack, and must be transferred to the unspecialized code that will evaluate `c` before the operand stack is cleared by the thrown exception. What makes this process tricky is that the number and types of values on the operand stack vary across deoptimization points; therefore we must have access to the stack size and type information at each deoptimization point in order to correctly transfer state. Unfortunately, managed VMs do not provide the ability to reflect on the operand stack during runtime.

We solve this problem by using compile-time[3] validation of the generated intermediate representation. To achieve this, the code generator is combined with a bytecode verifier which verifies the generated code line-by-line during code generation (as opposed to the normal order, which completely generates the code and then validates it). The benefit

[3]Throughout this chapter, "compile" refers to generation of the typed bytecode of the underlying VM from the dynamic language being implemented on that VM. This should not be confused with the native code generation that happens at the VM level.

of this approach is that, in order to verify type safety, the code verifier maintains a shadow stack of value types present in the operand stack at any program point. The code generator can take advantage of this information during code generation, whereas it could not do this if the validator waited until after generation is complete.

This approach has two benefits beyond enabling correct state transfer. First, it enables runtime validation of the VM intermediate bytecode generated by the dynamic language runtime, which aids the language implementor in detecting compiler errors early rather than waiting until the code is actually run and the underlying VM gives an "Invalid IR" message. Secondly, there are certain unusual circumstances where the values on the operand stack cannot be transferred correctly to the unspecialized code, and hence deoptimization is not feasible (this is discussed further in Section 3.4.3). The code verifier will detect such circumstances and mark the code as un-optimizable.

3.4 Deoptimization for MCJS

This section concretely explains the algorithm for deoptimization that we have implemented in MCJS, a JavaScript engine implemented on top of the Common Language Runtime (CLR). MCJS performs type feedback based type inference to generate type specialized code. The type inference algorithm implemented in MCJS is described in Chapter 2. The explanation in this section uses a running example given in Figure 3.3: a JavaScript function `foo` that takes an argument `a` which the example assumes is always an integer value.

The function `foo` is initially interpreted by the MCJS runtime. When `foo` becomes warm, it is compiled by the fast compiler into CIL[4] bytecode. This fast compilation also: (1) uses the code verifier to detect the types of values present on the operand stack

[4]Common Intermediate Language, a typed bytecode IR used by the CLR.

```
function foo(a)
{
  var b = 10;
  return a + b + global;
}
```

Figure 3.3: Running example in JavaScript.

for each potential deoptimization point, and determines for each point if deoptimization is feasible;[5] and (2) instruments the code to collect type profiling information. Finally, if `foo` becomes hot then it is re-compiled by the optimizing compiler into (1) a type-specialized CIL bytecode version based on the collected profile information; and (2) an unspecialized subroutine-threaded version used by the deoptimizer to recover from unexpected types.

The remaining subsections expand on the optimizing compiler pass: we explain first the subroutine-threaded code generator and then the specialized code generator that handles deoptimization.

3.4.1 Subroutine-Threaded Interpreter

When a hot function is compiled, the optimizing compiler first generates subroutine-threaded code for that function before generating type-specialized code. The order is important, because the specialized code needs to have pointers to the appropriate subroutines for each potential deoptimization point. Table 3.1 shows the subroutines that are generated for the example function in Figure 3.3. The only possible place for deoptimization (assuming `a` is always an integer) is if the type of `global` changes during some subsequent execution of `foo`. Thus, subroutine 5 is the subroutine that the runtime will jump to if deoptimization occurs. Since the subroutine-threaded interpreter executes a

[5] This is discussed further in Section 3.4.3.

Index	Subroutine Name	Expression	Operand Stack
0	WriteIndentifier	b = 10	[]
1	ReadIdentifier	a	[a]
2	ReadIdentifier	b	[a, b]
3	AddExpression	a+b	[a+b]
4	ReadIdentifier	global	[a+b, global]
5	AddExpression	a+b+global	[a+b+global]
6	Return	return	[]

Table 3.1: Subroutines generated for a subroutine-threaded interpreter corresponding to the example in Figure 3.3. Subroutine 5 is the unspecialized code where control is transferred by the deoptimizer if `global` contains an unexpected type during the specialized code evaluation.

sequence of subroutines for each operation in the function, it is important to maintain an explicit stack that mimics the operand stack across the subroutines. MCJS implements this operand stack in the `callFrame` data structure described in Figure 3.6. The operand stack generated by subroutine 4 needs to be reconstructed by the deoptimizer before jumping into subroutine 5. The method to do so is explained below.

3.4.2 Specialized Code Generator

The generated type-specialized code contains deoptimization hooks at each potential deoptimization point. These hooks are filled in with the addresses of the appropriate subroutines generated as per the above description. In the example, the deoptimization code in the guard around `global` is compiled with a pointer to subroutine 5. Figure 3.4 shows the CIL code that is generated for the expression `a+b+global`.

It remains to explain how a deoptimization point transfers control to the unspecialized code subroutine while maintaining the current program state. We first explain how control is transferred from the specialized code into the unspecialized code, and then we explain how program state is transferred along with the control.

```
...
0055   ldloc a
0056   ldloc b
0057   call Int32 Binary.Add:Run (Int32, Int32)

... ;  TYPE CHECK
... ;  Load the global variable
0071   dup
0072   call int DValue:get_ValueType()
007b   ldc.i4 9 ; 9 = observed type = Int32
0080   beq fast ; jump to fast path

... ;  DEOPTIMIZATION CODE
... ;  Update the profiler with observed type.
... ;  Transfer the operand stack to the
... ;  callFrame->stack data-structure.
... ;  Explained in Table 2.
00d4   ldc.i4 5 ; 5 is the index of the
                ; subroutine to jump into.
00db   throw GuardFailedException(Int32)

... ;  FAST PATH
fast   call Int32 DValue:AsInt32() ; Unboxing
00e5   call Int32 Binary.Add:Run(Int32, Int32)
... ;  Set the return value in the callFrame
00f4   ret

... ; CATCH BLOCK
... ; Store the current values of the local
... ; variables into the callFrame->symbols array.
... ; BlackList this function.
... ; Load the callFrame object that contains
... ; the updated stack and symbols.
... ; Load the subroutine index obtained from
... ; the exception value.
0147   ldc.i4 subroutineIndex
014c   call Void STInterp(Int32, CallFrame)
```

Figure 3.4: CIL code generated by the type-specializing code generator for the expression `a + b + global`.

```
void __foo(CallFrame *callFrame)
{
 int a, b;

 try {
  b = 10;
  a = callFrame->argument[0].ToInt();

  int _temp0 = a + b;
  DValue _temp1 = callFrame->getGlobal("global");

  /* TYPE CHECK */
  if (_temp1.type != Int) { // Int is the profiled type
    /* DEOPTIMIZATION CODE */
    /* Update the profiler with newly observed type */
    UpdateProfiler(global, Int);
    /* Capture the current values of _temp* */
    callFrame->stack.Enqueue(_temp1); // Enqueue(DValue);
    callFrame->stack.Enqueue(_temp0); // Enqueue(int);
    /* 5 is the pointer to the subroutine */
    throw new GuardFailureException(5);
  }
  else { // FAST PATH
    callFrame->retVal = DValue(_temp0 + _temp1.AsInt32());
    return;
  }
 }
 catch (GuardFailureException e) {
  /* Update the callFrame->symbols array with the
     current values of local variables */
  callFrame->symbols[symbolsIndex++] = DValue(a);
  callFrame->symbols[symbolsIndex]   = DValue(b);
  BlackList(this); // BlackList this function code.
  STInterp(e.subRoutineIndex, callFrame);
 }
}
```

Figure 3.5: C-like psuedocode that describes the generated CIL for the JavaScript code in Figure 3.3. The values pushed onto the stack are made explicit using _temp variables.

Control Transfer. The jump to the deoptimization code is implemented using the exception handling feature of the CLR. Each specialized method is wrapped in a try-catch block. Before a `GuardFailure` exception is thrown at a deoptimization point, the runtime updates the profiler with the new type that was observed, in order to improve the profiler's type information. The operand stack is then captured at the point the exception is thrown. The function locals, in contrast, are captured inside the catch block; this is because the operand stack is specific to a particular deoptimization point while the locals are common across all deoptimization points in the function. Capturing the values of the local variables in a single place avoids code duplication and reduces code bloat.

Once inside the catch block and with all local variables captured, the runtime must clean up and then transfer control to the appropriate subroutine. First, the runtime calls the `Blacklist` function which deletes the specialized code that had to be deoptimized and updates the function metadata with this information; this prevents the function from entering a cycle of specialization followed by deoptimization over and over again. Secondly, the runtime calls the appropriate subroutine whose pointer was passed inside the `GuardFailure` exception, passing it the updated `callFrame` data structure as explained below.

State Transfer. In MCJS, the `callFrame` data structure tracks the state of execution for the current function. It also holds a link to the scoping structure used to resolve the scope of the variables used in the function. Figure 3.6 shows the definition of `callFrame`. MCJS uses the `callFrame` object to transfer program state from the specialized code to the unspecialized subroutine-threaded code. The two relevant fields are `symbols`, which holds the values of the function's local variables at the deoptimization point, and `stack`, which holds the operand stack at the deoptimization point.

The `symbols` field is computed inside the specialized function's catch block, as

```
struct CallFrame {
  // Arguments passed to the function
  DValueArray arguments;
  // Return value of the function
  DValue retVal;

  // Fields and functions to track the scope
  // and other bookkeeping.
  Scope currentScope;
  Scope parentScope;

  // Fields below are only used by the
  // subroutine-threaded interpreter.
  // symbols array is used to store the values of
  // local variables at the deoptimization point.
  DValueArray symbols;

  // stack array is used to capture the state of
  // operand stack at the deoptimization point.
  DValueArray stack;
}
```

Figure 3.6: The `callFrame` data structure which tracks the state of execution for the current function.

explained above. This is straightforward for the MCJS implementation because the runtime maintains a list of local symbols; the catch block merely iterates over this list and copies the values into the `callFrame.symbols` field.

The `stack` field must be computed separately for each deoptimization point. For each point, the type and number of values that need to be pushed onto the stack are different. The code generator used to generate the specialized CIL code uses the bytecode verifier to track this information. The verifier is reponsible for inferring and checking type information, which means that it already needs to know the required information. We simply piggyback on the verifier to determine what code to emit for enqueueing the operand stack values at each deoptimization point. The verifier maintains a data

structure called the TypeStack which holds the types of values inside the operand stack at each program point. At each deoptimization point, we record the current TypeStack and emit code to enqueue the operand stack values onto `callFrame.stack`. Each value is wrapped inside a DValue before being enqueued. Because in CIL value types are not subtypes of the Object type, the runtime cannot use a generic `Enqueue(Object)` method to enqueue the values which is why we need the verifier's TypeStack information.

Table 3.2 shows how the state transfer code is generated for the example in Figure 3.3. Maintaining a TypeStack during code generation helps to determine which variation of `Enqueue` has to be called to enqueue the value in the top of the operand stack to `callFrame.stack`. In the example, while enqueuing `global` from the operand stack, the top of the TypeStack is referred for the appropriate type. Since the type of `global` is DValue, the CIL code to call `Enqueue(DValue)` is emitted by the code generator. Similarly, a call to `Enqueue(Int32)` is emitted to capture the value of `a + b` from the operand stack.

3.4.3 Limitations

Our deoptimization technique assumes that all values present on the operand stack at a deoptimization point are subtypes of DValue. If so, then all of the values are easily convertable to value types used in the JavaScript runtime. However, there are rare cases where this assumption is not true. Some optimizations, such as polymorphic inline caches, store the `map` or `class` of an object in the operand stack of the CLR. If a deoptimization is triggered at this point, state transfer is not possible because `map` cannot be converted to a DValue and stored in `callFrame.stack`.

Fortunately, it is easy to detect this ahead of time during code generation. During the fast compilation phase which translates warm functions to CIL bytecode and instruments

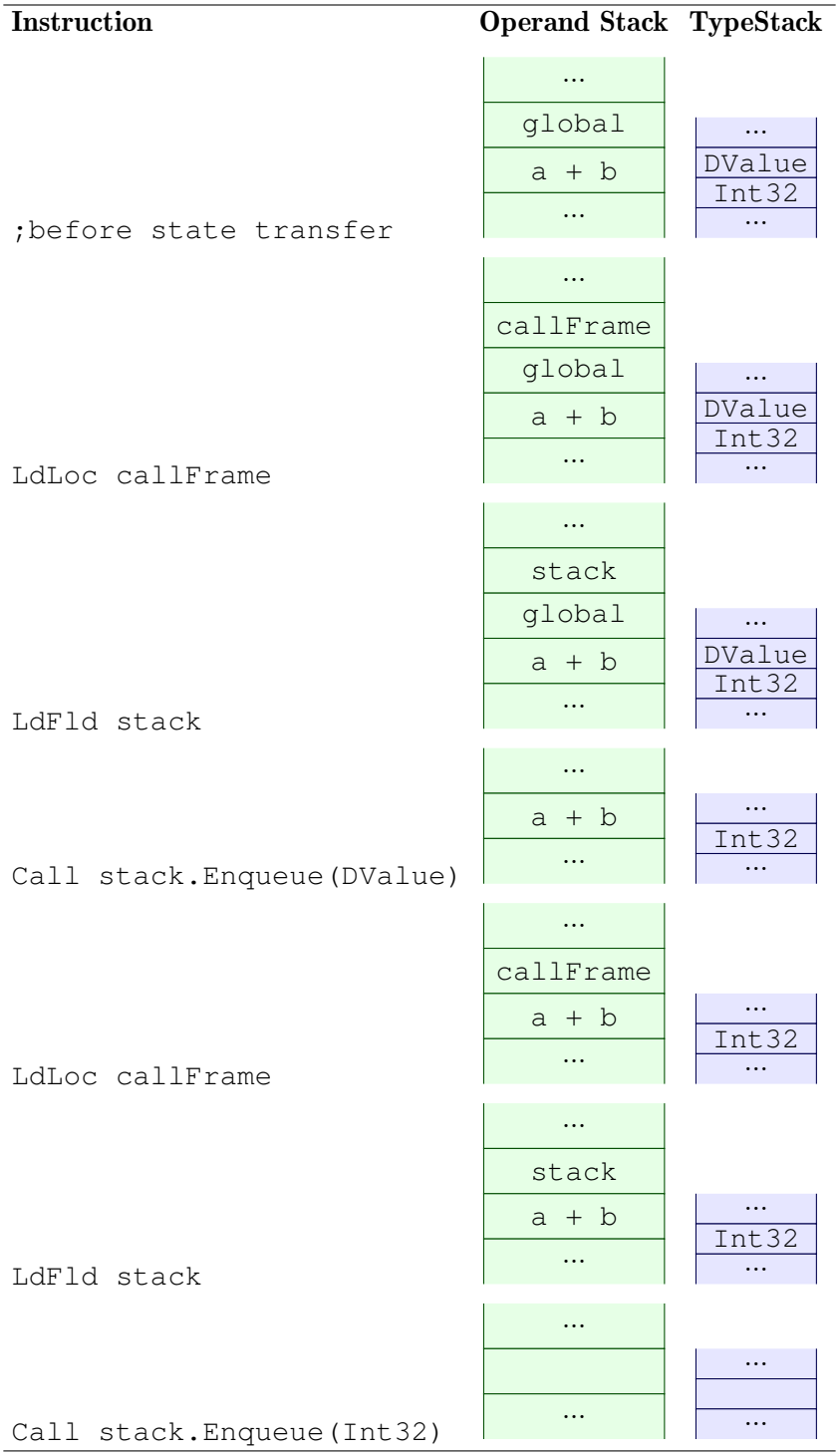

Table 3.2: The different steps taken when popping values from the operand stack.

the code with type profiling hooks, the types of the values in the operand stack are tracked by the code verifier as previously described. For every deoptimization point, the type stack is checked to see whether it contains values that cannot be converted to DValue. If so, then the function is marked as non-optimizable. The profile hooks are removed and the function is compiled directly to CIL without any type feedback-based type specialization. Our evaluation shows that this circumstance rarely happens.

3.5 Evaluation

We evaluate our deoptimization technique on MCJS using the standard JavaScript benchmark suites Sunspider [22] and V8 [23] [6]. Because the Sunspider benchmarks run for a short duration of time (average of 180ms), each benchmark was wrapped in a 20× loop. We also evaluate our technique on real-world long-running web applications from the JS1k [25] website. Due to the unstable nature of IronJS, we selected only the benchmarks that IronJS was able to execute without any problem. The JS1k benchmarks are described in the Table 3.3.

Experimental Setup. We perform our experiments on a machine with two 6-core 1.9 GHz Intel Xeon processors with 32GB of RAM, running the Ubuntu 12.04.3 Linux OS and Mono v3.2.3. We used the latest version of IronJS, v0.2.1.0 from its Github repository [10].

Calculating Speedups. To calculate execution times, each of the benchmarks is run eleven times and the average execution time of the last ten executions is recorded.

[6] MCJS and IronJS do not implement typed arrays. Therefore, we not evaluate our implementation on Octane benchmarks.

Benchmark	**Type**
breakout.js	Game
chopper.js	Game, Animation
colorfulPointer.js	Utility, Animation
conways.js	Animation, Algorithm
flyingWindows.js	Animation, Utility
loadingSpinner.js	Utility
sierpinskiGasket.js	Algorithm
analogClock.js	Utility
halloweenAnim.js	Animation
growingGrass.js	Animation
kaboom.js	Game
mandelbrot.js	Animation, Algorithm
plasma.js	Animation
primesAnim,js	Algorithm
springPond.js	Algorithm, Animation
tetris.js	Game
waveGraph.js	Algorithm, Utility

Table 3.3: Table describing JS1k web applications used as benchmarks.

Configurations. Speedup numbers were collected for the following five configurations.

- MCJS without type feedback-based type specialization (the base configuration against which results for other configurations are normalized).
- MCJS with type specialization using the standard fast path + slow path recovery mechanism (MCJS_FS).
- MCJS with type specialization using the deoptimization recovery mechanism, i.e., our technique (MCJS_D).
- MCJS with optimal type specialization (MCJS_OPT) as described below.
- IronJS in its default configuration.

The optimal type specialization configuration means that code is type-specialized but there is no deoptimization or any other recovery mechanism; this is unsound, but provides a maximal speedup due to type specialization against which we can compare

our technique and the cost of deoptimization. IronJS is implemented on top of DLR [49] which mimics the fast path + slow path approach to optimizing type specializable code, hence we use it to show that MCJS is not a strawman JavaScript implementation.

3.5.1 Speedups

Figure 3.7 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the Sunspider benchmark suite. The approaches without a local slow path (i.e., MCJS_D and MCJS_OPT) perform significantly better than the fast path + slow path approaches implemented in MCJS and IronJS. The MCJS_OPT configuration does not emit any deoptimization code and the runtime exits when any deoptimization should occur, which is why the 3d-cube.js benchmark sees a speedup of 0× for the MCJS_OPT configuration.

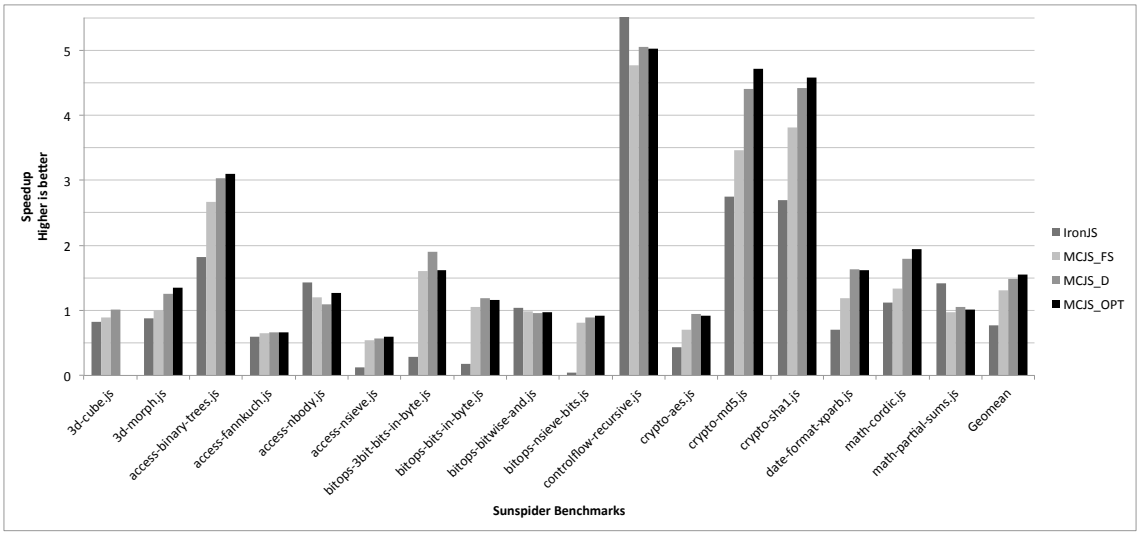

Figure 3.7: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the Sunspider benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

On an average (geomean) MCJS_D, MCJS_FS, and IronJS are 1.5×, 1.31×, and 0.77× faster than the base configuration, respectively. On comparing the execution times of

MCJS_D against MCJS_FS and IronJS, we see an average speedup (geomean) of 1.14× and 1.97× respectively. An important observation is that for a few of the benchmarks like access-fannkuch.js, access-nbody.js, access-nsieve.js, bitops-bitwise-and.js, etc, the runtime does not benefit from type feedback-based type specialization. This is because these benchmarks are relatively small and execute for a very short period of time (average of 237.2ms). For these benchmarks profiling overhead is not amortized over time.

Figure 3.8 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the V8 benchmark suite. We selected the benchmarks for which IronJS executed without crashing. Following a similar trend as the Sunspider benchmarks, MCJS_D, MCJS_FS, and IronJS are 2.13×, 1.74×, and 1.21× faster than the MCJS base configuration. On comparing the execution times of MCJS_D against MCJS_FS and IronJS, we see an average (geomean) speedup of 1.22× and 1.75× respectively. Excluding regexp.js (for which MCJS spends most of the time executing the inefficient regexp library code) and splay.js (which is a benchmark designed for stressing the garbage collection of the engine rather than the runtime performance), MCJS_D consistently performs better than all other configurations.

The JS1k benchmarks represent a diverse set of applications including games, utilities, algorithms, and animations. We manually modified the JavaScript code to eliminate or substitute code that interacted with the browser DOM. We substituted `setTimeOut` and `setInterval` functions with JavaScript functions that execute the passed-in function in a loop for a considerable number of times. For the benchmarks that require user interactions like mouse clicks, the user events were simulated using a fixed set of event objects embedded in the code. These applications run for a relatively long duration with the average execution time for the base configuration being 10.66 seconds.

Figure 3.9 shows the speedups achieved by the type specializing configurations with respect to the MCJS base configuration for the JS1k web application benchmark suite.

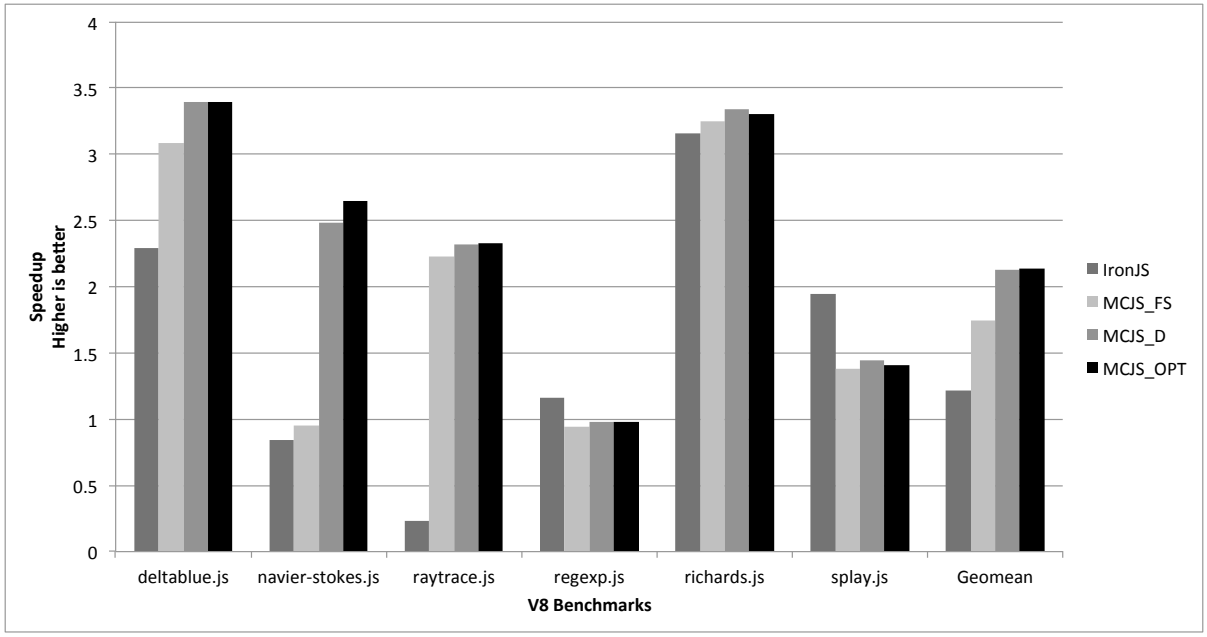

Figure 3.8: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the V8 benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

As expected, MCJS_D, MCJS_FS and IronJS are 1.76×, 1.5×, and 0.94× times faster than the MCJS base configuration. Similar to the other benchmark suites, on comparing the execution times of MCJS_D with MCJS_FS and IronJS, we see an average speedup of 1.18× and 1.87× respectively.

Speedup vs. V8: MCJS achieves on an average about 75% of the V8 engine performance on the Sunspider benchmarks. The speedup is significantly lower for few of the benchmarks in V8 benchmark suite. This is mainly because the regexp and string library implementations of MCJS (which are based on CLR's implementations) are very slow. Those affects dominate performance for those benchmarks, rather than anything due to recovery. However, this is not a fair comparison because V8 implements both the recovery mechanisms as part of its compilers along with many other optimizations, making it very difficult to tease out and isolate the effect of each of the recovery mechanisms.

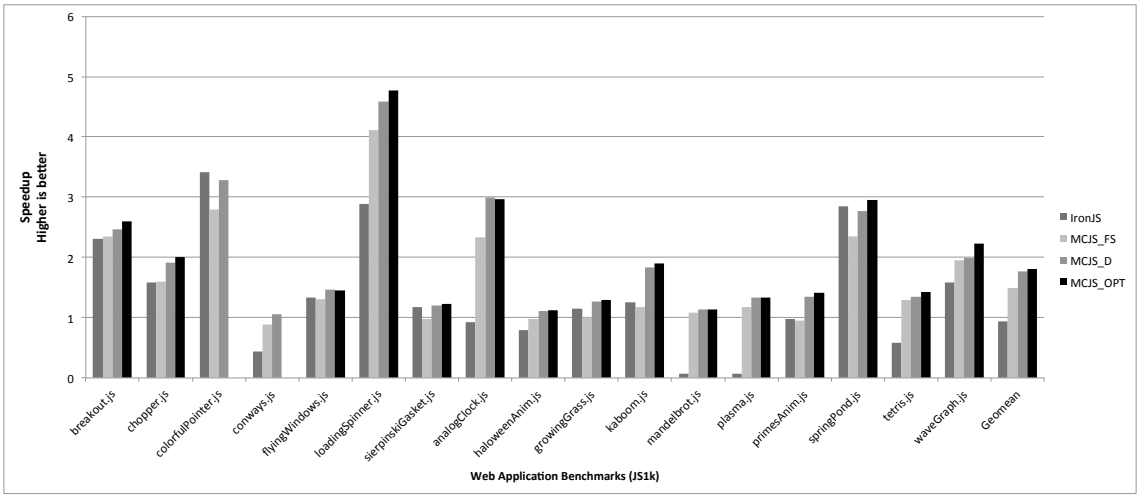

Figure 3.9: This figure shows the speedup numbers for various configurations of MCJS and IronJS for the web application benchmark suite. FS = fast path + slow path, D = deoptimization, OPT = optimal.

3.5.2 Effect of Deoptimization

Deoptimization is a rare occurrence and it is observed only during the execution of the 3d-cube.js, colorfulPointer.js, and conways.js benchmarks. The speedup numbers for these benchmarks indicate that the overhead of the actual deoptimization process is negligible.

There are two ways of measuring the effect of the deoptimization code. First, we compare the speedup achieved by the MCJS_D and MCJS_OPT configurations. Figures 3.7, 3.8, and 3.9 indicate that the runtime overhead of the deoptimization code is negligible.

Secondly, we compare the size of the extra code that is generated to achieve deoptimization for each of the benchmarks. Figure 3.10 shows the comparison on code size of MCJS_D and MCJS_FS with respect to MCJS_OPT. Though the amount of code that is generated in MCJS_D is approx. 30% higher compared to the MCJS_OPT configuration, the impact on performance is negligible. This is because most of the extra code that is generated is to enable deoptimization. This deoptimization code is rarely ever executed

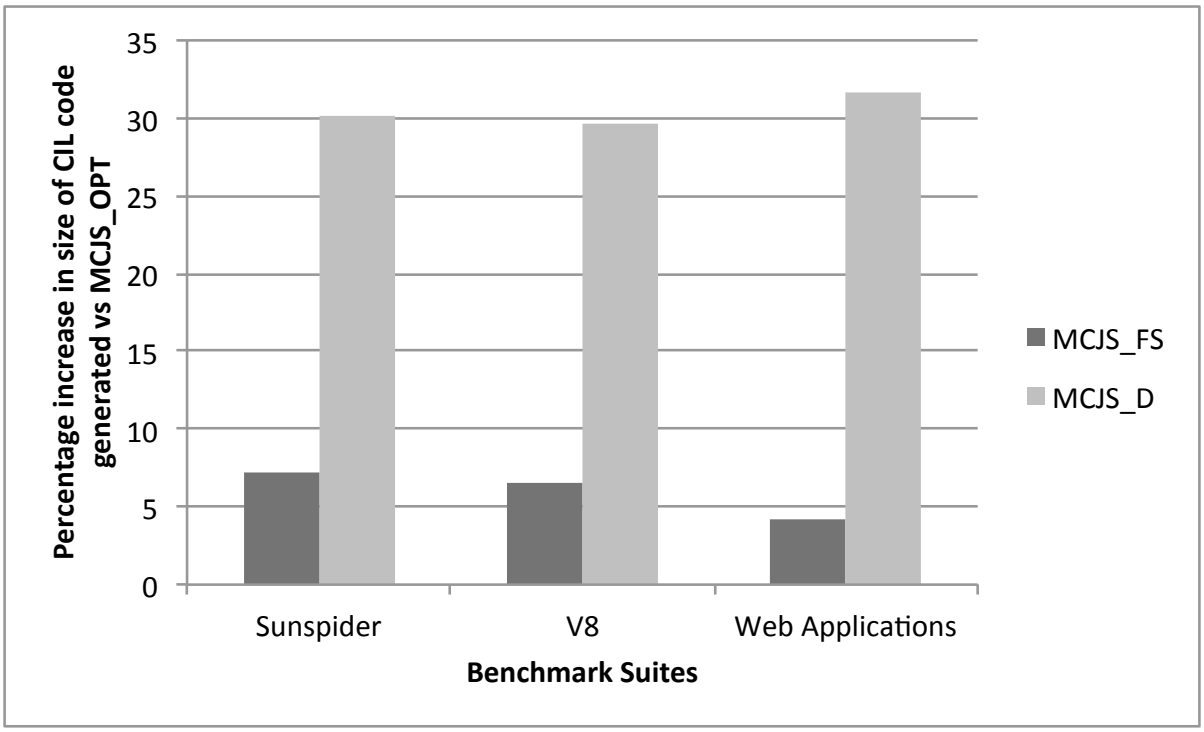

Figure 3.10: This figure shows the percentage increase in CIL code generated for MCJS_FS and MCJS_D in comparison to MCJS_OPT. FS = Fast + Slow Path, D = Fast Path with Deoptimization, and OPT = Fast Path with No Deoptimization.

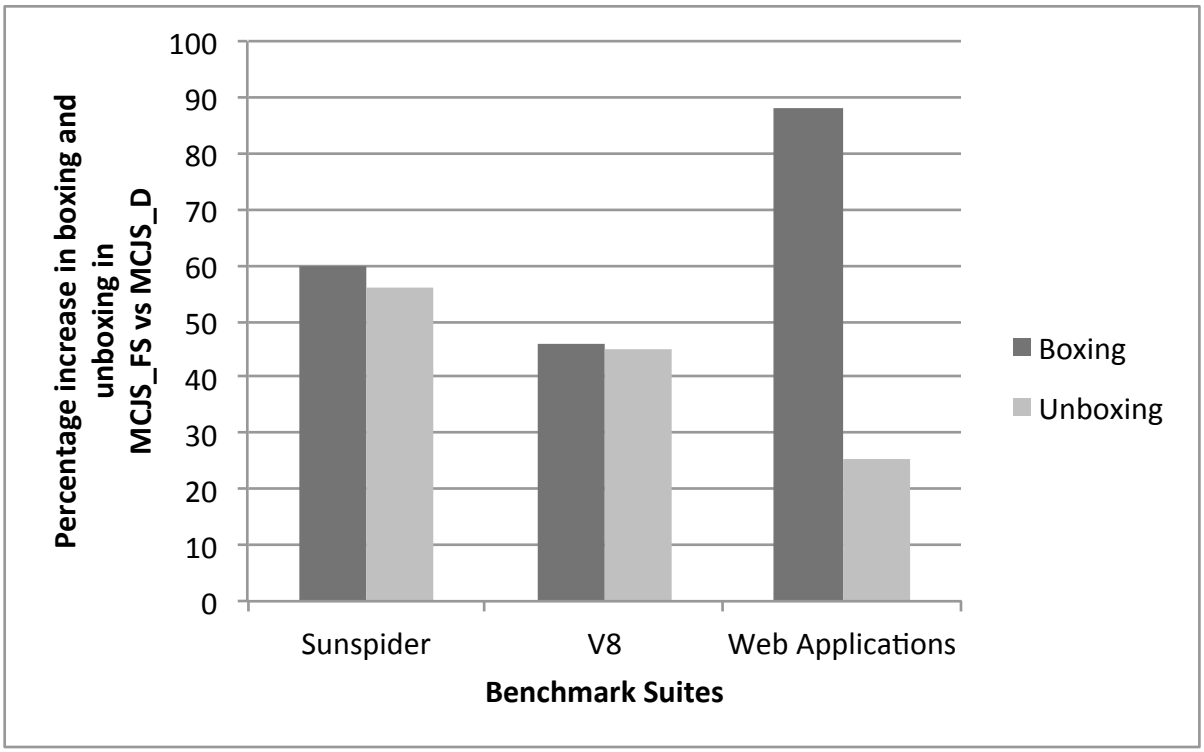

Figure 3.11: This figure shows the percentage increase in boxing and unboxing in MCJS_FS in comparison to MCJS_D. FS = fast path + slow path and D = deoptimization.

for most of the benchmarks.

Another important metric used while comparing two implementations is the memory consumption. The amount of data captured in the stackFrame data-structure is very minimal; the operand stack and values associated with local variables are usually a few bytes in size. Therefore, the stackFrame data-structure has little to no impact on memory when we consider a managed runtime system.

3.5.3 Boxing/Unboxing

The amount of boxing and unboxing of DValues performed during the execution of the benchmarks is a major cause of overhead for the MCJS_FS configuration. Figure 3.11

shows the percentage increase in boxing and unboxing performed in MCJS_FS configuration when compared to the MCJS_D configuration for each of the benchmark suites. As expected, MCJS_FS performs more boxing and unboxing of values when compared to MCJS_D across all benchmark suites.

An important observation is that the percentage of boxing for web applications is significantly higher compared to other benchmarks. This is because the number of variables that are typed in the MCJS_D configuration is significantly higher compared to the number of local variables which are typed in the MCJS_FS configuration. This means that for the MCJS_FS configuration almost all the values need to be boxed before assigning them to the variable.

3.5.4 Non-optimizable Code

In some benchmarks, the deoptimization approach is not possible because some values present in the operand stack cannot be converted to DValues, as explained in Section 3.4.3. Among all benchmarks from the various benchmark suites that were executed, the runtime was not able to generate deoptimization code for only 35 out of 448 functions that were classified as hot. This shows that the deoptimization approach is viable for type specialization on top of VMs.

3.6 Related Work

Both the fast path + slow path and the deoptimization approaches for type specialization have been used in various dynamic language runtimes implemented natively (rather than on top of a VM). The baseline compilers of popular JavaScript engines V8 [29, 30] and SpiderMonkey [14] use the fast path + slow path approach for initial compilation of JavaScript functions to native code. Once a function becomes hot, the optimizing

compilers for both of these engines generate type-specialized code with deoptimization hooks. If the types used for specialization change during execution, the runtime performs deoptimization by initiating long jumps to deoptimization routines in the compiled machine code. Language runtimes written on top of typed, stack-based runtimes cannot implement such a deoptimization technique because of the typed nature of the IR and the runtime type safety guarantees enforced by the VM.

TraceMonkey [50], PyPy [34], and LuaJIT [51] are popular tracing JIT compilers. Deoptimization is a common approach to use in runtimes with trace-based compilers. These traces span across function boundaries and are compiled to native code with deoptimization hooks. Implementing such a trace-based compiler on top of a VM is very complicated, especially from the perspective of deoptimization.

Brunthaler et al [52, 53] describe a purely interpretative optimization technique called *Quickening* implemented in CPython runtime. Quickening involves rewriting generic instructions to optimized alternatives based on the runtime information. This is analogous to the fast-path + slow-path approach of optimization. Quickening with deoptimization can be an alternative to the existing approach of optimization.

Hackett et al [8] describe an approach of combining type inference with type feedback to generate type specialized code. This approach uses recompilation approach instead of classic deoptimization technique to bail out whenever the type related assumptions do not hold anymore in the compiled code. Their approach tracks the type of values held by a variable or object field, and recompile all the type specialized code to generic version when the new types are observed.

Dynamic Language Runtime (DLR) [49] based language implementations like IronJS [10], IronRuby, and IronPython compile the program written in the dynamic language into DLR's *ExpressionTrees*. DLR performs the optimizations and native code generation required for the runtime. DLR employs polymorphic inline caches to specialize any

operation observed during execution, which is analogous to the fast path + slow path approach of type specialization. As observed in Section 3.5, such an optimization does not always result in good performance when compared to the deoptimization approach.

Ishizaki et al [54] implement a dynamically-typed language runtime by modifying a statically-typed language compiler. Their approach to type specialization modifies the compiler to generate fast path + slow path code for arithmetic, logical, and comparison operators. Similarly to the MCJS original fast path + slow path approach, their approach also has to deal with incessant boxing and unboxing of values.

Duboscq et al [55] describe a way of inserting and coalescing deoptimization points in the IR of the Graal VM. This technique is orthogonal to and complementary to our own. In our approach, the deoptimization points are determined while generating the subroutine threaded code for the interpreter. Our implementation can benefit from the techniques like coalescing and movement of deoptimziation points described in their paper.

On-stack replacement (OSR) is a deoptimization / reoptimization strategy that has been explored and implemented in language runtimes to enable speculative optimizations [43, 44, 45, 46] and to enable debugging of optimized code [56]. Hölzle et al [56] implement deoptimization for the SELF programing language for debugging optimized code. The main focus of this work is to maintain the mapping from optimized compiled code to source code. As the authors have complete control over the underlying VM, such deoptimization is relatively easy to implement when compared to our implementation which does not modify the underlying VM. Fink et al [43] describe an on-stack replacement strategy for deoptimization implemented in JikesRVM. As described in the paper, capturing the state of the execution is straightforward given the access to the JVM scope descriptor object of the executing code. Our implementation is not straightforward due to the fact that part of the state that needs to be transferred resides in the underlying

operand stack which is not easily accessible by any code currently executing in the VM. Soman et al [44] present a new general-purpose OSR technique on JikesRVM which is decoupled from the optimization performed by the runtime. Similar to this approach our deoptimization technique is also general-purpose. Applying the current deoptimization technique to other optimizations would involve minor modifications to the subroutine threaded interpreter to indicate the expected points of deoptimization specific to that optimization.

3.7 Conclusion

Deoptimization is a recovery mechanism which allows the runtime to bail out of type specialized code when type assumptions are violated, capture the state of current execution and continue execution form an equivalent point in a unspecialized code. This chapter proposes a novel deoptimization based type-specialized code generation for a dynamic language runtime implemented on top of a typed, stack-based virtual machine. Our approach does not require any modification to the underlying virtual machine. Our implementation uses the exception handling feature offered by the underlying VM to perform deoptimization. Just using exception handling feature to jump into unspecialized code is not enough because throwing an exception clears the operand stack of the VM. The operand stack is an important part of state that needs to be transferred during deoptimization. We leverage the shadow type stack maintained by the bytecode verifier, which verifies the validity of the code generated during its generation, to safely transfer the values in the operand stack to the unspecialized code.

We implement our proposed technique in MCJS, a research JavaScript engine running on top of the Mono runtime. We evaluate our implementation against the fast path + slow path approach implemented in MCJS and IronJS. Our results show that deoptimization

approach is on an average (geomean) 1.16× and 1.88× faster than fast path + slow path approach implemented in MCJS and IronJS respectively on Sunspider, V8 and web application benchmark suites.

Our implementation is generic and can be extended to enable other optimizations like function inlining. A few minor modifications to the existing approach are required to implement it in a sound manner. Currently, the location of deoptimization is determined by the placement of type checking guards. This needs to be extended to incorporate possible places where function inlining is possible in the code.

Chapter 4

Server-Side Type Profiling

Modern JavaScript engines optimize hot functions using a JIT compiler along with type information gathered by an online profiler. However, the profiler's information can be unsound and when unexpected types are encountered the engine must recover using an expensive mechanism called *deoptimization*. In this chapter we describe a method to significantly reduce the number of deoptimizations observed by client-side JavaScript engines by using ahead-of-time profiling on the server-side. Unlike previous work on ahead-of-time profiling for statically-typed languages such as Java [57, 58] our technique must operate on a dynamically-typed language, which significantly changes the required insights and methods to make the technique effective. We implement our proposed technique using the SpiderMonkey JavaScript engine, and we evaluate our implementation using three different kinds of benchmarks: the industry-standard Octane benchmark suite, a set of JavaScript physics engines, and a set of real-world websites from the Membench50 benchmark suite. We show that using ahead-of-time profiling provides significant performance benefits over the baseline vanilla SpiderMonkey engine.

4.1 Introduction

Modern JavaScript engines have multi-tier execution architectures with sophisticated optimizing JIT compilers. Like optimizing JIT compilers for statically-typed languages (e.g., the JVM [59] and CLR [60]), JavaScript JIT compilers optimize based on profile information collected during execution. But unlike those other JITs, the collected profile information for JavaScript is of a different nature involving heuristic type information that is not guaranteed to be correct. When a function is optimized using profile-based type assumptions, there is a chance that those assumptions will not hold in the future. The JavaScript JIT compiler will optimize a hot function based on the types observed during the previous executions of the function. In the future, if new, unexpected types are encountered during execution of the optimized code, the JavaScript engine must employ a recovery mechanism called *deoptimization* to guarantee correctness. This recovery mechanism is a heavy-weight, expensive process that can severely impede the engine's performance.

In this chapter we propose a technique that uses ahead-of-time type profiling on the webserver side in order to determine type and hotness information for a JavaScript program; that information is sent to the web browser client as commented annotations in the JavaScript code, and the client uses that information to reduce the number of deoptimizations during execution. Client JavaScript engines that are aware of the ahead-of-time profiling information can take advantage of it, while client engines that are not aware of it can safely ignore it. The intent of this technique is *not* to reduce profiling or compilation overhead (which turn out to be mostly insignificant), but rather to reduce the number of deoptimizations during program execution and also to enable more aggressive and earlier optimization of functions without having to fear increased deoptimizations.

A naïve approach to ahead-of-time type profiling for JavaScript would simply observe

the execution of the program on some set of inputs and (1) mark all functions that become hot sometime during the execution, so that they can be optimized immediately instead of waiting; and (2) remember all types seen during the execution of those hot functions, so that the optimized versions will not have to be deoptimized due to type changes. However, it turns out that this naïve approach would significantly *degrade* performance on the client and would also create program annotations potentially orders of magnitude larger than necessary. We explain the reasons behind this observation and our key insights that allow ahead-of-time profiling to be both practical and effective.

Previous work for statically-typed language JIT compilers has proposed using ahead-of-time profiling, as discussed in Section 4.2. However, JavaScript provides a new setting that requires new techniques and insights. We show that for JavaScript: deoptimization is an important performance concern; ahead-of-time profiling can provide significant performance benefits by avoiding deoptimization; and the annotation comments in the JavaScript code sent from the server increase code size by only a small fraction. The specific contributions of this work are:

- We describe a method for ahead-of-time profiling of JavaScript programs to collect type and hotness information. We identify the key kinds of information and places to collect that information that provides the most benefit for optimization without requiring excessive annotations on the program code being sent over the network.

- We describe a method for JavaScript engines to take advantage of the ahead-of-time profiling information to reduce deoptimizations and to more aggressively optimize functions without incurring increased deoptimizations.

- We evaluate our ideas using Mozilla's JavaScript engine SpiderMonkey. Our experiments show that our technique is beneficial for both load-time and long-running JavaScript applications, as represented by the Membench50 load-time benchmark

suite, the industry-standard Octane performance benchmark suite, and a set of open-source JavaScript physics engines. We measure the performance using three different criteria: execution time for Octane benchmarks, frames per second (FPS) for the JavaScript physics engines, and reductions in number of deoptimizations for the Membench50 benchmarks. Our evaluation shows a maximum speedup of 29% and an average speedup of 13.1% for Octane benchmarks, a maximum improvement of 7.5% and an average improvement of 6.75% in the FPS values for JavaScript physics engines, and an average 33.04% reduction in deoptimizations for the Membench50 benchmarks.

The rest of the chapter is organized as follows. Section 4.2 describes related work on optimizing JIT compilers. Section 4.3 provides background information on the JavaScript language and on modern JavaScript engine architectures. Section 4.4 describes the concepts behind our technique. Section 4.5 describes our evaluation methodology and results, and Section 4.6 concludes.

4.2 Related Work

Our work builds on decades of research into optimizing JIT compilers, such as Self [45, 56], Java HotSpot VM [46], Jalepeno [61], PyPy [34], Google's V8 engine [29], Mozilla's SpiderMonkey [14], and WebKit's JavaScriptCore [62]. We review the most relevant of that related work below.

4.2.1 Ahead-of-Time Profiling

Ahead-of-time profiling for the purpose of optimizing a JIT compiler is not a new idea, but previous efforts have focused on statically-typed languages such as Java and

C#. JavaScript, a dynamically-typed language, provides a new setting that dramatically changes the required insights and techniques. In particular, the most important optimization performed by a JavaScript JIT compiler is type specialization based on unsound heuristics such as online type profiling. Because type specialization is unsound, the engine must be able to deoptimize the specialized code when it encounters unanticipated types. Deoptimization is an important cause of performance loss and is the main target of our technique, unlike any of the previous ahead-of-time profiling techniques described below. We do in addition follow previous work in using ahead-of-time profiling to detect hot functions that can then be compiled early. However, our new setting also influences this existing technique in new ways because merely detecting *hotness* is insufficient—we must also ensure that the hot function is *type stable* for early compilation to have any benefit, otherwise deoptimization is likely to happen. We now describe the previous work on ahead-of-time profiling for JIT compilation, which all target statically-typed languages.

Krintz and Calder [58, 63] describe an approach to identify hot functions and hot callsites in Java programs using analysis information collected offline. This information is used by the JIT compiler to guide its optimization heuristic. Our approach is similar to their approach of using offline data to guide online optimizations. Unlike their approach, our offline profiler collects type information and deoptimization information in addition to hot functions and hot callsite information. The type information is important for a dynamically-typed language such as JavaScript because most of the optimizations that are performed in the optimizing compiler depend on stable type information. Deoptimization information helps to figure out possible places where deoptimizations occur in the hot functions and the reasons why deoptimization has happened. This information helps the optimizing compiler to make better decisions while compiling those hot functions.

Arnold et al. [57] describe an Java virtual machine architecture that uses a cross-run profile repository to improve performance. The main idea described in that paper is to capture the profile data at the end of the execution of the program instead of discarding it after every run. This collective profile information is used to guide the selective optimization of functions based on metrics like future use. A key idea of that work is to address the *compilation time* vs. *future execution time* trade-off inherent in single-threaded engines that interleave execution and JIT compilation. Modern JavaScript engines employ concurrent JIT compilation, and so compilation time is generally not as important an issue. Also, we take advantage of the client/server infrastructure inherent in the world-wide web to do the ahead-of-time profiling on the server side and send the resulting information to the client for it to take advantage of, rather than doing profiling in the client itself.

4.2.2 Type Annotations for JavaScript

Developers and researchers have created several typed variants of JavaScript. These variants are either restricted subsets of the full JavaScript language or do not allow the types to be used by the JIT compiler for optimization.

The JavaScript dialect `asm.js` [64] is a strict subset of JavaScript that is intended to be generated by compilation to JavaScript from some statically-typed language such as C. It indicates the types of variables and operations based on subtle syntactic hints and a `"use asm"` prologue directive. Though this enables the JavaScript engine to perform ahead-of-time compilation and faster execution, the `asm.js` syntax is very restrictive and is not suitable for writing modern webpages. It is designed to be an intermediate representation for porting applications written in statically-typed languages into the web browser. In contrast, our approach deals with already existing JavaScript programs and

handles the entire JavaScript language.

Flow [65] is a static type checker for JavaScript that allows type annotations in the syntax. These annotations are used by the compiler to type-check the code for correctness. An optimizing JIT compiler cannot make use of these annotations because the annotations are erased during the translation of Flow code to JavaScript. This is also true of Google's Closure compiler [66], which allows type annotations in the JavaScript code, and of TypeScript [67], a typed superset of JavaScript.

4.2.3 JavaScript Engine Optimizations

Guckert et al. [68] show that persistent caching of compiled JavaScript code across visits to the same webpage helps reduce compilation time by up to 94% in some cases. However, because the optimizing compiler usually runs in a separate parallel thread compilation time is not much of a concern in modern JavaScript engines. In addition, this technique does not translate well to a setting where the server is responsible for collecting information and sending it over the network to clients, due to the large size of the compiled code and its specificity to a particular architecture.

Oh and Moon [69] describe another client-side optimization technique targeting load-time JavaScript code (i.e., JavaScript code executed when a webpage is loaded by the browser). This technique caches snapshots of the heap objects that are generated during the load time; the snapshots are serialized during caching and then deserialized when the page is reloaded. This approach uses significant amounts of storage space and does not translate well to server-side profiling.

4.3 Background

In this section we describe background on the JavaScript language and modern JavaScript engines required in order to understand the key concepts discussed in this chapter.

4.3.1 The JavaScript Language

JavaScript is an imperative, dynamically-typed scripting language with objects, prototype-based inheritance, higher-order functions, and exceptions. Objects are the fundamental data structure in the language. Object properties (the JavaScript name for object fields) are arbitrary strings and can be dynamically inserted into and deleted from objects during execution. Because property names are just strings, a JavaScript program can compute a string value during execution and use it as a property name in order to access an object's existing property or to insert a new property. A form of runtime reflection can be used for object introspection in order to iterate over the properties currently held in an object. Even functions and arrays are just different kinds of objects, and can be treated in the same way as other objects, e.g., inserting and deleting arbitrary properties. JavaScript is designed to be resilient even in the face of nonsensical actions such as accessing a property of a non-object (i.e., a primitive value) or adding two functions together; such cases are handled using implicit type conversions and default behaviors in order to continue execution as much as possible without raising an exception.

4.3.2 JavaScript Engine Architecture

Modern JavaScript engines rely heavily on profiling and JIT compilation for performance. The JIT compiler relies on type information gathered by the profiler in order to enable effective optimizations in the face of JavaScript's inherent dynamism. Type in-

formation includes not just the primitive kinds of values (`number`, `boolean`, `string`, `object`, `undefined`, and `null`), but in addition information about *object shape*, i.e., a list of object properties and their offsets in the object. Because properties can be arbitrarily added to and deleted from an object, object shapes can change frequently during execution.

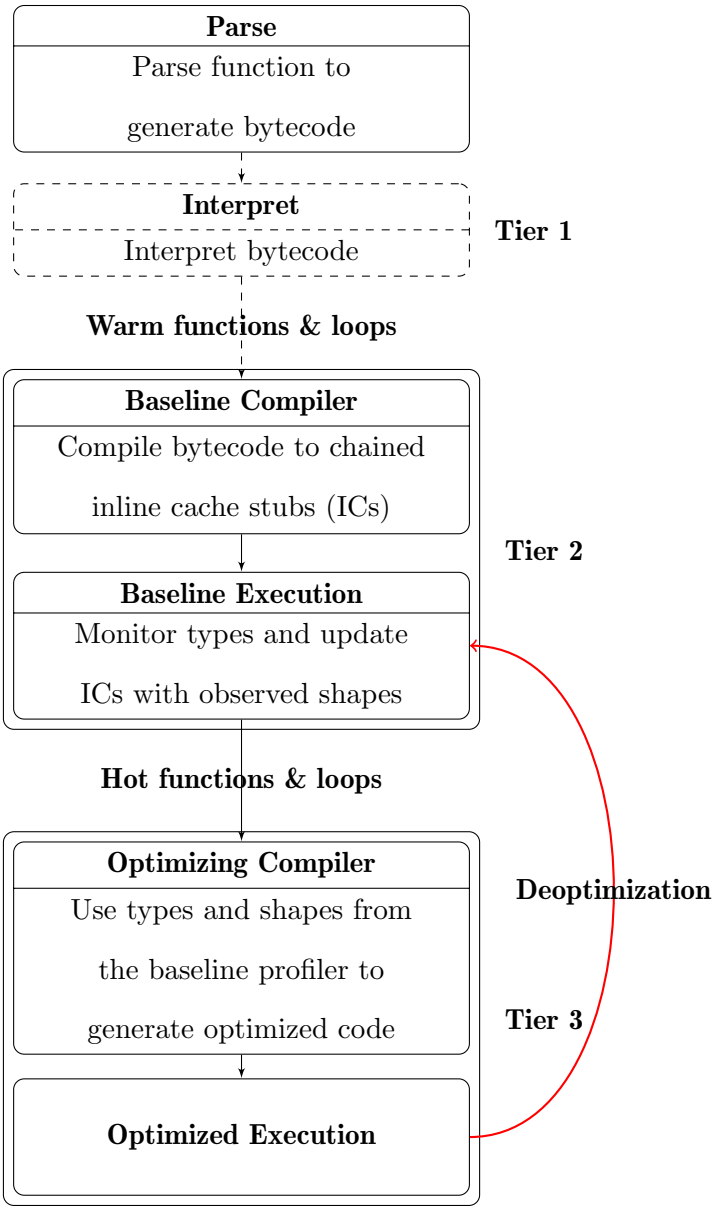

Figure 4.1: Flow graph showing different phases of execution in a generic JavaScript engine. The interpretation phase, represented by dashed lines, is an optional phase in JavaScript engines like Google's V8.

Figure 4.1 shows a typical multi-tier architecture for a generic JavaScript engine, based on the designs of existing production JavaScript engines such as Google's V8 [30], Mozilla's Spidermonkey [14], and WebKit's JavaScriptCore [62]. These tiers operate at the granularity of individual functions.

Tier 1. The first tier of execution is a fast interpreter for parser-generated bytecode. The interpreter is used to ensure quick response times during execution of the JavaScript program. For example, SpiderMonkey's bytecode interpreter and JavaScriptCore's LLInt execute functions for the first 10 and 6 times they are called, respectively. Once the given threshold is reached, the function is considered warm. Not all engines use this first tier; for example, the V8 engine skips the interpreter and goes straight to tier 2.

Tier 2. The second tier of execution is a baseline compiler that compiles the bytecode to assembly code as quickly as possible, with minimal optimization. The baseline compiler also inserts instrumentation into the compiled code to collect profiling information. The profile information that is collected by the baseline compiler includes the types of variables and of object properties and the shapes of objects whose properties are accessed/modified during function execution. The baseline code is executed many times before a function is considered hot, e.g., SpiderMonkey typically executes the baseline compiled code one thousand times before moving to the next tier [70]. This large threshold is intended to help ensure that the type information gathered by the profiler is stable and hopefully will not change in the future (if it does change then *deoptimization* will be triggered, discussed in Section 4.3.3). This threshold may vary based on other factors such as whether a function contains back-edges which are frequently visited or whether a function has been deoptimized earlier.

Tier 3. The third tier of execution is an optimizing compiler that compiles hot functions based on profile information collected in the previous runs of the function, i.e., the baseline compiled code. The profile information may be invalidated by future calls to the function being optimized (e.g., the types may change), therefore the optimized code also contains *guards* that check the assumptions under which the code was com-

piled. If those guards are violated then deoptimization will happen, moving execution from the optimized code back to the baseline compiled code from tier 2. The optimizing compiler performs various optimizations such as loop invariant code motion, common subexpression elimination, guard hoisting, function inlining, and polymorphic cache inlining to speed up the execution of the function. The optimizing compiler is relatively slow compared to the interpreter and baseline compiler, therefore modern JavaScript engines adopt a concurrent compilation strategy. Using this strategy, the time taken for compilation is not a big concern for performance because it is not in the critical path for program execution.

4.3.3 Type Specialization and Deoptimization

Many of the most effective optimizations performed by the optimizing compiler are based on *type specialization*. For example, consider the expression "`a + b`". In the general case (without type specialization), variables `a` and `b` will refer to boxed values residing in the heap that are tagged to specify the types of those values; these are called *dynamic values*. To perform the + operation, the code must unbox `a` and `b`, determine their respective types based on their tags (requiring a series of branch instructions), perform any type conversions necessary, perform the operation, then box the resulting value along with its type tag. This process must be used for *all* operations on dynamic values, significantly slowing the execution time.

If, however, the compiler has reason to believe that `a` and `b` are (almost) always of certain types based on the profile information collected by the baseline compiled code, then it can specialize the optimized function to those types. It does not need to use dynamic values for `a` and `b`, it can use unboxed values instead; it does not need to check type tags to determine what operation to perform for +, it can directly use the operation

pre-determined by the known types of `a` and `b`, and it can optimize the emitted code based on this knowledge. This is an example of type specialization, and it is one of the most effective means available for improving execution time of dynamically-typed languages. Another example of type specialization takes advantage of object shape information to efficiently access object properties, e.g., using polymorphic inline caches [71].

The essential problem is that the profile information on types and object shapes is necessarily unsound—observed types during profiling do not guarantee what the types will be in future executions. Deoptimization is the recovery mechanism the engine uses when current types do not match the assumptions used when optimizing the function in tier 3. The engine does not discard the baseline compiled code from tier 2 when it generates the optimized code in tier 3. Instead, for each guard point where type information is checked and may be invalidated, the engine maintains a mapping from the optimized code to the equivalent point in the baseline compiled code. When a guard fails, execution stops at that guard in the optimized code and resumes at the equivalent point in the baseline compiled code, which is not type specialized and hence can handle any possible type. Deoptimization is an inherently expensive operation, and reducing the number of deoptimizations is a primary goal when optimizing engine performance.

Deoptimizations can be classified into different categories depending on their exact cause. A **type-based** deoptimization is caused by attempting to use a value that has a different primitive type than expected (`number`, `boolean`, `string`, `object`, `undefined`, or `null`). A **shape-based** deoptimization is caused by attempting to access an object that has a different shape than expected (i.e., the property offsets are potentially different). These are the two kinds of deoptimizations that we target in this chapter.

Other kinds of deoptimizations are caused by speculative, optimistic assumptions made by the optimizing compiler that may not be valid. For example, because arrays are just objects in JavaScript, array elements in the middle of the array can be deleted

leaving a "hole" in the array. The optimizing compiler assumes that there are no holes in an array. Numbers in JavaScript are doubles, but the optimizing compiler assumes that they are integers for added performance. Computed property accesses (i.e., computing an arbitrary string and using it as a property name) can be anything, but the optimizing compiler assumes that it will be a property actually in the object being accessed. For all of these assumptions the compiler must emit a guard to check that assumption in case it is not true, and trigger a deoptimization if it is not. We do not focus on these kinds of deoptimizations in this work, but they are an interesting target for future work.

4.4 Ahead-of-Time Type Profiling

In this section we describe our technique for performing ahead-of-time profiling on the server-side and for taking advantage of that profile information on the client-side. Figure 4.2 gives a high-level overview of our technique's flow. The server will profile the JavaScript program in two phases to collect profile information. This usually happens during the feature testing or regression testing phase of the application. Instead of a regular browser, the developers use a lightly modified version of the browser (as described later in this section) while performing the manual or automated regression testing, in order to collect the profile information. The application is then annotated with this profile information. The advantage of this approach is that whenever the application is updated, a new profile can be captured using the existing test suite and a new version of the annotated program can be created. The annotated version is then sent over the network to the client on request; the client can take advantage of that information if it is aware of the ahead-of-time profiling, or safely ignore that information if it is not.

We now describe in detail the server's ahead-of-time profiling technique and the insights required to collect the most useful information, and then the modifications required

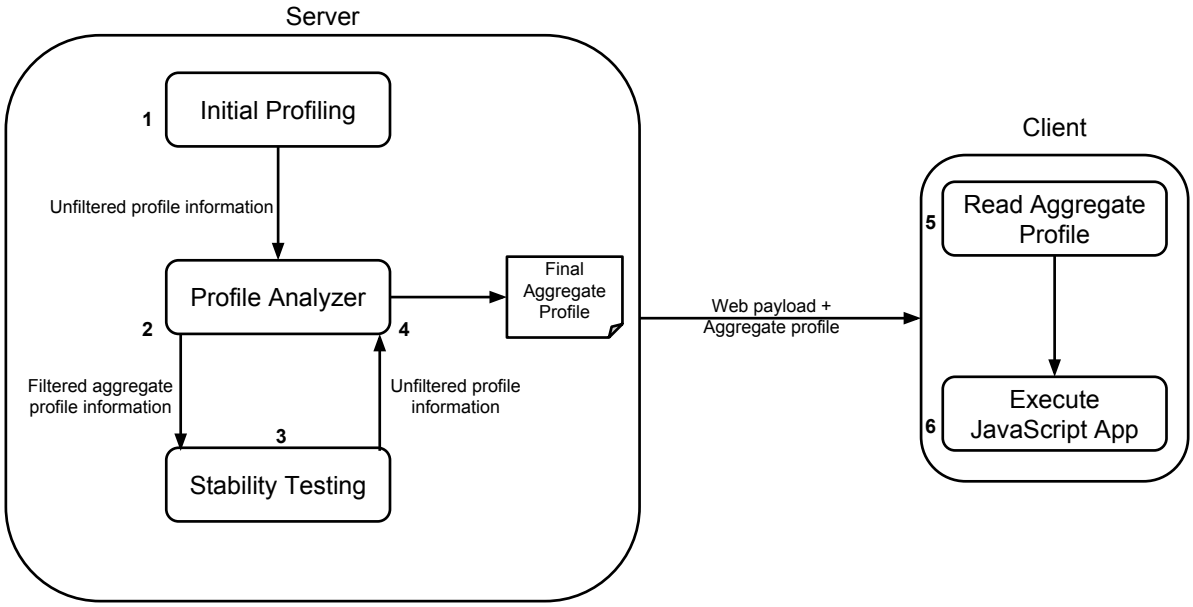

Figure 4.2: Profiling and execution setup. (1) The server performs initial profiling of the JavaScript application to generate unfiltered profile information. (2) The profile analyzer filters out relevant information and invokes (3) stability testing. (4) The profile analyzer collects the profile information from (3) and (1) to generate an aggregate profile. When the client requests the web application, aggregate profile information is sent along with the webpage. The client (5) reads the aggregate offline profile information and combines it with online profile information during the (6) execution of the application.

of the client to take advantage of the profile information. To clarify a potential point of confusion: we distinguish between the engine's standard runtime profiler used to collect information for the optimizing compiler (the *online profiler*) and our ahead-of-time profiler (the *offline profiler*) that collects the information gathered by the online profiler and preserves it past the end of the program's execution.

4.4.1 Server-Side Profiling

We concentrate our efforts specifically on type-based and shape-based deoptimizations. We currently ignore the other kinds of deoptimizations discussed in Section 4.3, though they may be interesting targets for future work. The ahead-of-time profiler op-

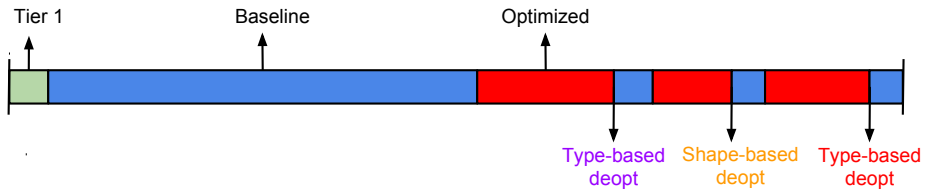

Figure 4.3: Timeline showing the execution of a single function within a JavaScript program in the server.

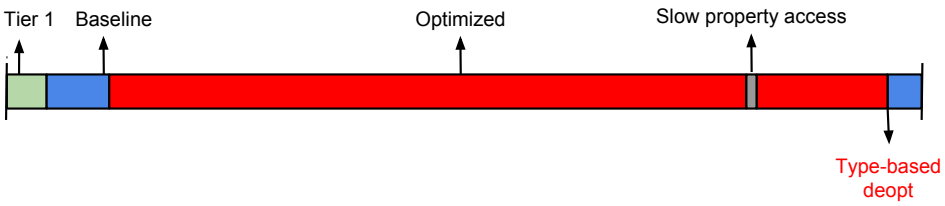

Figure 4.4: Timeline showing the execution of a single function within a JavaScript program in the client.

erates in two phases: the **initial phase** and the **stability testing phase**. We begin with the initial phase.

Initial Profiling Phase

Consider Figure 4.3, which shows an example execution timeline for a single function within the program being profiled, where time is measured in number of function invocations. The function starts off in tier 1 (the green portion). Once the function becomes warm it goes through the baseline compiler and starts executing the baseline compiled code (the blue portion). During this time the online profiler is collecting information about types and object shapes. Once the function becomes hot it goes through the optimizing compiler and starts executing the optimized compiled code (the red portion). If the function must be deoptimized, for example, by a changing type or object shape, then the function reverts back to the baseline compiled code (and again is being profiled by

the online profiler). If the function subsequently becomes hot again then it is recompiled by the optimizing compiler. The optimizing compiler may inline function calls as part of its optimizations, thus the optimized function may incorporate additional code over the baseline version.

By modifying the online profiler to save its information to an external file (a simple change to an API that all modern browsers already provide for accessing the online profile information), the offline profiler has access to each function's timeline, including whether the function ever became hot and all of the profile information about types and object shapes used by the function, as well as the number of deoptimizations that happened, where they happened, and why they happened.

Naïvely, one could attempt to optimize the program by marking each hot function and providing all of the collected type and shape information. The client could then immediately compile all hot functions (with parallel compilation to avoid load-time latency), using the provided type and shape information. Ideally this would allow the client to use optimized code right away while avoiding all (type- and shape-based) deoptimizations found during offline profiling (in Figure 4.3, all of the blue portions would be replaced by red portions). However, this approach does not work for three reasons.

Reason 1. Tracking shape information requires a lot of data, significantly increasing the amount of annotations that need to be sent over the network. Also, there is a large cost for the client in terms of memory and time, because the client needs to keep track of the executing program's object shape information in order to take advantage of the information from the ahead-of-time profiler. Our preliminary experiments show that tracking shape information in the client takes at least several megabytes of memory and increases client execution time by an average of 27%. We want to be able to reduce shape-based deoptimizations without incurring this overhead. Therefore, rather than have the

offline profiler record actual shape information we instead have it record the program points at which shape-based deoptimization happened during the ahead-of-time profiled execution. This information is sent to the client instead of the exact shape information; we explain how the client uses this program point information in the next subsection.

Reason 2. There is a very common coding idiom used by JavaScript programmers that uses the primitive JavaScript values `null` or `undefined` as a sentinel value to signal the end of some iteration. For example, think of a list of integers that is terminated by a `null` value to signal the end of the list. A variable `x` holding the value of the current position in the list would be an integer up until the point that sentinel value is reached, then variable `x` changes type to `null` instead. This type change then triggers a type-based deoptimization. We could eliminate that deoptimization by compiling the function at the beginning knowing that `x` could be an integer or `null`. However, the problem is that this newly optimized function would actually run *slower* than the original optimized function with the deoptimization, because the original compiled function could perform many optimizations relying on `x` being an integer and is thus much faster than the newly compiled code which must account for `x` being either an integer or `null`. The original deoptimization is slow, but happens once and only at the end of the function's lifetime. The end result is that while the newly optimized function avoids the deoptimization, it is in aggregate slower than the original optimized function plus the deoptimization.

For this reason, our offline profiler ignores the type information from the last type-based deoptimization in the last optimized execution of the function being profiled (in Figure 4.3, this would be the last type-based deoptimization in the figure). We assume that this deoptimization is from the coding idiom described above, and therefore we allow that last type-based deoptimization to remain.

Other deoptimization patterns can also occur due to very rare unexpected types or

shape modifications during the execution of the application. Our technique does not consider this as a special case and records the unexpected type in the log. When this type is used in the client side to optimize the hot function, the optimizing compile might generate sub-optimal code. Therefore, there is a tradeoff between cost of executing sub-optimal code without deoptimization using offline profile information versus the collective cost of deoptimization, the cost of profiling after deoptimization, and the cost of executing sub-optimal code after the function is regarded hot again. In most cases we found that ignoring the last deoptimization was sufficient for better performance.

Reason 3. Some functions are inherently type unstable and will consistently be deoptimized no matter how much profile information is saved (often because of other kinds of deoptimizations rather than type- and shape-based deoptimizations). Optimizing these functions will result in a net loss in performance because of the constant deoptimization. We set a threshold value for number of deoptimizations (of all kinds, not just type- and shape-based) and mark all functions that exceed this threshold as non-optimizable in the program's profile annotations.

Stability Testing Phase

The amount of annotations added to a program as comments by the offline profiler increases the size of the program, and hence provides a cost in terms of network bandwidth when sending the program from the server to the client. We would like to minimize that cost as much as possible. The purpose of the stability testing phase is to figure out which program annotations may have unnecessary information, which we can then drop to minimize the annotation size. This phase is solely to optimize the size of the program annotations, it does not affect the client-side optimizations (either positively or negatively).

Recall that the threshold for making a function hot and sending it through the optimizing compiler is significantly higher than it might be in a statically-typed language (on the order of 1,000 invocations) in order to make it more likely that the baseline profiler has seen all of the relevant type information before optimization, thus reducing the chances of deoptimization. This threshold is very conservative because the cost of those deoptimizations is so high. The idea of the stability testing phase is to detect functions that stabilize much earlier than this threshold; for those type-stable functions we can omit the type information from the profiler's annotations (while still marking them as hot). At the client we drastically reduce the amount of time the baseline profiler is run before optimizing a function that is marked hot by the offline profiler, but still leave enough time that these type-stable functions have all of the necessary type information collected by the client engine's online profiler. In other words, we are dropping the type annotations for the type-stable functions to save space, then reconstituting them on the client side via the normal online profiling. The stability testing phase identifies the type-stable functions where we can be sure that the online profiler will get all of the necessary type information even though we are reducing the amount of time it has to profile those functions.

We define a potential type-stable function as one that, in the initial phase, was marked as hot but had no deoptimizations (except perhaps a final type-based deoptimization per the coding idiom described earlier). We detect type-stable functions empirically by rerunning the same program on the same input as for the initial phase, but taking all of the potential type-stable functions and initializing their hotness counter to a high value (rather than the normal zero), but *without* using any of the type information gathered by the initial phase. Any potential type-stable function that still does not have any deoptimizations is considered type-stable and their type annotations are removed from the program.

4.4.2 Client-Side Optimization

When the client receives a program containing annotations from our ahead-of-time profiler (given as code comments), it strips them from the program when that program is read into the engine's memory and stores them in an object we call the Oracle. The Oracle stores the profile information indexed by function.

When a function is first loaded, the client engine consults the Oracle to determine if it is a hot function. If so, the function's hotness counter is initialized to a high value (rather than zero) so that it will be quickly passed to the optimizing compiler. If, on the other hand, the profile information indicates that this function is too type unstable, then the client engine marks it as unoptimizable so that it will never be passed to the optimizing compiler. These two mechanisms (the hotness counter and the unoptimizable mark) are already present in all modern JavaScript engines, and thus engines using our technique need only minor modifications to take advantage of the Oracle's information.

When a function is being compiled by the optimizing compiler, the Oracle is again consulted to gather the profiled type information. The optimizing compiler already consults the online profiler to gather type constraints in order to perform type inference; it is a simple change to have it also gather type constraints from the Oracle as well. The optimizing compiler also already uses shape information from the online profiler to inline accesses to objects (i.e., to use offset information to jump directly to a property rather than using a hash table). It is again a simple modification to have the compiler consult the Oracle to determine if a particular object access triggered a shape-based deoptimization during ahead-of-time profiling, and if so to avoid inlining the access. By avoiding this optimization we eliminate the possibility of shape-based deoptimizations at this point; while the lack of optimization slows down the code, the benefit of avoiding deoptimization provides a net gain in performance.

Consider Figure 4.4. This figure represents an example timeline on the client side for the same function as Figure 4.3 (which represented the function's execution on the server-side during profiling). We see that there is still a warmup phase, but that the function is optimized much earlier than before. The first two deoptimizations are removed, but at the expense of an unoptimized object property access to eliminate the shape-based deoptimization. Finally, the last type-based deoptimization still remains to account for the common JavaScript coding idiom described previously.

4.5 Evaluation

To evaluate the benefits of our ahead-of-time profiling technique, we implement it using Mozilla's production-quality JavaScript engine SpiderMonkey and test it on three different benchmark suites:

- **Octane**, the industry-standard JavaScript performance benchmark suite [72].
- **Physics**, a set of open-source JavaScript physics engines for web games [73, 74, 75, 76].
- **Membench50**, a benchmark suite consisting of real-world websites that heavily use JavaScript [77].

Note that throughout this section, "deoptimizations" refers specifically to type- and shape-based deoptimizations.

We run our experiments on an 8-core Intel i7-4790 machine with 32GB RAM running Fedora 20 Heisenbug as the profiling server and another machine with the same configuration as the client. Below, we first give details on the modifications we made to the SpiderMonkey engine for our implementation and then describe the results for each of the three benchmark suites in turn.

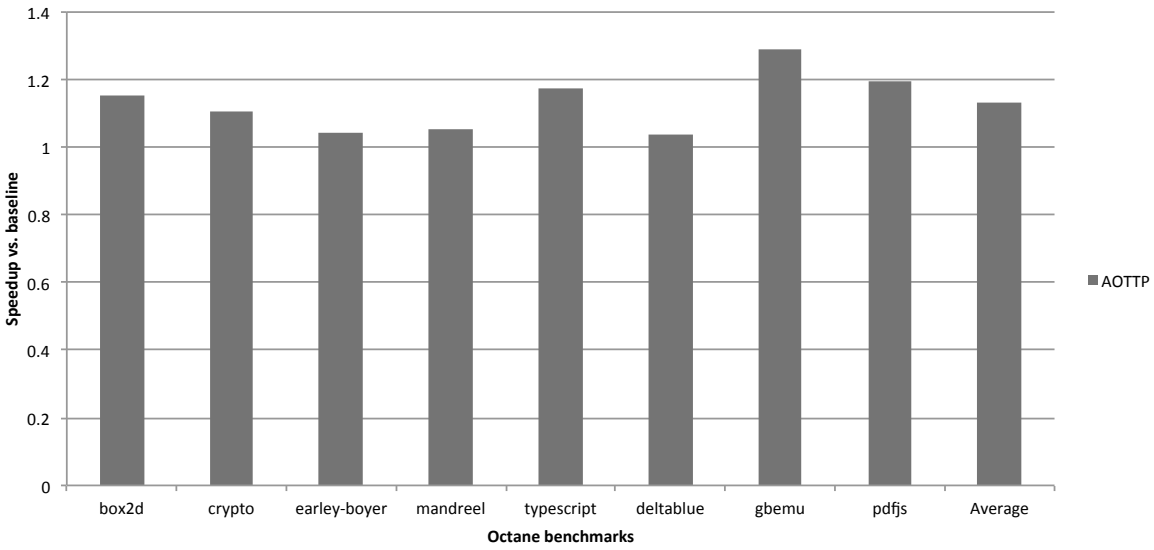

Figure 4.5: Average speedup of Ahead-Of-Time Type Profiling (AOTTP) versus the baseline. Higher is better.

4.5.1 SpiderMonkey Modifications

We modify SpiderMonkey version 224982 from the mozilla-central repository [78] as the profiling engine and the client engine. The modified engines are available as an anonymized download. [1] We use an unmodified SpiderMonkey of the same version as the baseline to compare against. The specific modifications and heuristics that we use for the server and the client are as follows.

Server. The profiling engine is modified to log the following three classes of information along with the location in the source where they are observed. The location is defined by {*file name, line number, column number*} in the source code.

- All type-based and shape-based deoptimizations that occur during the execution of the program.

[1] https://dl.dropboxusercontent.com/u/206469/dls15.zip

Table 4.1: Number of hot functions, the total number of deoptimizations observed during the baseline and AOTTP approach, and the percent reduction in number of type and shape-based deoptimizations.

Benchmarks	# hot funcs	Baseline deopts	AOTTP deopts	% reduction in deopts
box2d	254	15	5	66.66
crypto	55	12	4	66.66
earley-boyer	69	5	2	60
mandreel	71	0	0	–
typescript	324	73	49	32.87
deltablue	66	1	1	0
gbemu	171	20	17	15
pdfjs	93	25	15	40
Average	**137.88**	**18.85**	**11.62**	**40.17**

Table 4.2: Program annotation size, the benchmark size without annotations, and the percent size overhead when adding the annotations to the program

Benchmarks	Profile size (kB)	Benchmark size (kB)	% overhead in size
box2d	33.9	374.01	9.08
crypto	6.57	63.91	9.97
earley-boyer	6.04	211.10	2.81
mandreel	5.61	5016.26	0.11
typescript	51.99	1254.68	4.11
deltablue	5.14	41.58	12.37
gbemu	20.22	531.99	3.80
pdfjs	16.08	1482.06	1.06
Average	**18.08**	**1121.9**	**5.41**

- Any newly observed types that trigger type-based deoptimizations.
- Hot method and loop compilations that are performed by the IonMonkey optimizing compiler.

We use the SpiderMonkey default of 10 iterations as the warmness threshold and 1,000 iterations as the hotness threshold. Any function that is deoptimized more than 10 times is considered a type-unstable function.

For our prototype implementation we send the profile information as a separate text file which is read by the client, rather than embedding it in the original JavaScript

program. Embedding the annotations and stripping them out is trivial, but keeping them separate is more convenient for running experiments.

Client. The client is modified to read the program annotations into an oracle object. The oracle is consulted when the *JSScript* object is first created in the client engine. The warmup counter for a hot function is initialized to 950 instead of 0; thus hot functions will be compiled after 50 executions of the baseline compiled code rather than 1000.

4.5.2 Ahead-of-Time Profiling Cost

There is a small cost on the server-side to do the ahead-of-time profiling, though this cost is negligible. The profiler must run the program twice (once for each profiling phase) and write out the online profiler's information to disk. That information is then run through the profile analyzer to parse and collate the provided information in order to create the program annotations. This analysis process takes from a few milliseconds to a few seconds over all of our benchmarks.

4.5.3 Octane Benchmark Suite

The Octane benchmark suite is the industry standard benchmark suite used to measure the performance of JavaScript engines. Because our technique applies only to JIT compiled code, we consider a subset of Octane benchmarks which run for a reasonable amount of time, have a significant amount of hot functions (a minimum of 50), and have deoptimizations. For the other benchmarks in the suite, ahead-of-time profiling can be avoided altogether. Benchmarks like splay and regexp focus on different parts of the engine like the garbage collector and the regular expression engine; the zlib benchmark is an `asm.js` benchmark testing the efficiency of a different compiler in the JavaScript en-

gine; and the code-load benchmark does not exercise the optimizing compiler. Therefore, we do not consider those benchmarks. Choosing a subset of benchmarks is justified for our approach because unlike other online compiler optimizations that are always "on", the offline profile information based optimization is optional and can be disabled for applications which do not show additional speedup.

Calculating Speedup. Octane benchmarks provide scores upon completion of individual benchmarks. The higher the score the better the performance. To calculate the speedup, we run each benchmark 22 times in different VM instances and compute the average score of the last 20 times.

Training Inputs. Octane benchmarks typically do not take in any user input. Only a few of them take specific inputs from the external world, e.g., pdf.js and typescript. For example, the typescript benchmark is a typescript compiler that compiles itself. We used the `jquery.ts` file (with minor modifications) from the TypeScriptSamples Github repository[2] as the training input instead of the typescript benchmark's regular input. For the rest of the benchmarks, we modify them with different parameters to generate the training inputs for our server. For example, the crypto benchmark was modified to encrypt and decrypt different strings and the box2d benchmark was initialized with different step parameters. In this way we ensured that the ahead-of-time profiling was always done on different inputs than the client-side performance evaluation.

Observations. Figure 4.5 shows the speedups obtained by our ahead-of-time type profiles (AOTTP) against the baseline implementation. The most significant improvement in the performance is seen for the gbemu benchmark where the AOTTP approach is 29% faster compared to the baseline. On average the AOTTP approach shows 13.1% im-

[2]https://github.com/Microsoft/TypeScriptSamples

provement over the baseline configuration. Given the fact that SpiderMonkey is already highly optimized, this speedup is considered a significant performance improvement by the SpiderMonkey development team.[3]

Table 4.1 shows the reduction in deoptimizations when our AOTTP approach is used versus the baseline configuration. Except for mandreel and deltablue benchmarks, our AOTTP approach ensures that a significant amount of deoptimizations are avoided. On average 40.17% of the deoptimizations are eliminated using the AOTTP approach. The mandreel benchmark is generated using the Mandreel C++ to JavaScript compiler. Therefore, mandreel does not show any kind of deoptimization during the execution and is type-stable throughout the execution.

Comparing the percentage of deoptimizations reduced shown in Table 4.1 with the speedups numbers from Figure 4.5, one distinct observation would be that higher percentage of reduction in deoptimizations does not always correspond to improvement in performance. This is because, not all deoptimizations are in the critical path of execution of the benchmark. Avoiding deoptimizations that occur in a function that makes up most of the execution time would help improve performance of the application better.

The space overhead for the program annotations is minimal. Table 4.2 shows the size of the type profiles compared against the size of the benchmarks. On an average, the profile size is only around 5.41% of the size of the benchmarks. The typescript benchmark produces the largest profile information among all of the octane benchmarks. It produces a 51.59kB annotation which is around 4.11% of the size of the benchmark. A major chunk of the space overhead is due to the program location information which is {*file name, line number, column number*}. It is possible to drastically reduce this overhead by annotating the profile information directly in the source code instead of having a separate file.

[3]Personal communication at #jsapi IRC channel

Table 4.3: Reduction in deoptimizations for JavaScript physics engine demo applications and the improvement in FPS values when using ahead-of-time profiling.

Benchmark	# hot funcs	Baseline deopts	AOTTP deopts	Reduction in deopts	FPS improvement
three.js:canvas ascii	99	4	3	25%	7.3%
three.js:canvas camera orthographic	71	1	1	0%	7.5%
pixi.js:3D balls	31	10	3	70%	5.9%
matter.js:multi-body collision	50	7	0	100%	6.5%
physics.js:multi-body-collision	75	33	30	9.09%	6.5%
Average	**65.2**	**11**	**7.4**	**40.82%**	**6.75%**

4.5.4 JavaScript Physics Engines

Table 4.4: Profile annotation overhead for the JavaScript physics engine demo applications.

Benchmark	Profile size (kB)	Benchmark size (kB)	Size overhead
three.js:canvas ascii	12.00	864.47	1.40%
three.js:canvas camera orthographic	8.33	846.32	1.00%
pixi.js:3D balls	3.86	225.75	1.50%
matter.js:multi-body collision	7.92	602	1.30%
physics.js:multi-body-collision	13.13	556	2.36%
Average	**9.05**	**618.9**	**1.52%**

There is no definitive way of measuring JavaScript performance when embedded in a browser, so we take two different approaches in this subsection and the next. Here we want to measure computation-heavy JavaScript code performance in a browser setting. We use four open-source JavaScript physics engine demos as our benchmarks and use frames-per-second (FPS) as our metric for evaluating performance. Our hypothesis is that ahead-of-time profiles will show an improvement in the FPS values earlier in the execution of the benchmarks, because our ahead-of-time approach allows the client to optimize hot functions much earlier during execution. We believe that these demo applications best capture the behavior of computation-heavy applications where the user expects good

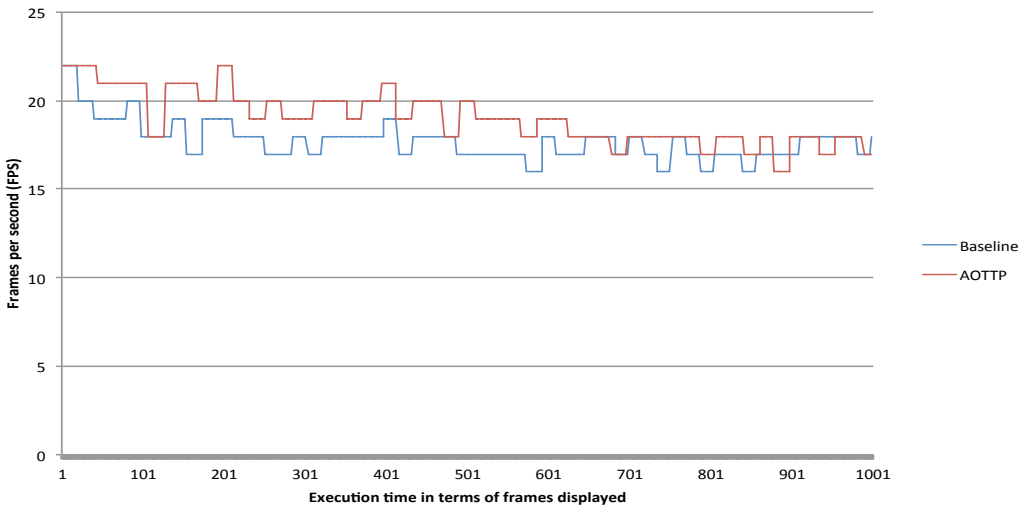

Figure 4.6: Frames per second (FPS) plots for three.js: canvas ascii JavaScript physics engine benchmark. The x axis represents execution times in terms of frames displayed. Higher is better

performance from the application from the time the application is launched.

Evaluation Setup. We evaluate 4 JavaScript physics engines: three.js [74], pixi.js [73], matter.js [75], and physics.js [76]. We use 2 demo applications from three.js and one each from pixi.js, matter.js, and physics.js, yielding a total of 5 physics engine benchmarks. These different engines have different ways of calculating FPS values. The three.js and pixi.js applications emit FPS values for every frame that is generated. Therefore, the x axis in Figures 4.6, 4.7, and 4.8 represent execution times in terms of frames displayed. The matter.js and physics.js applications emit FPS values every second. Therefore, in Figures 4.9 and 4.10 the x axis represent execution time in terms of seconds.

The benchmarks are inherently random and generate random collisions, patterns, and movements of objects for every invocation of the program. Therefore, our training input is guaranteed to be different compared to the evaluation input.

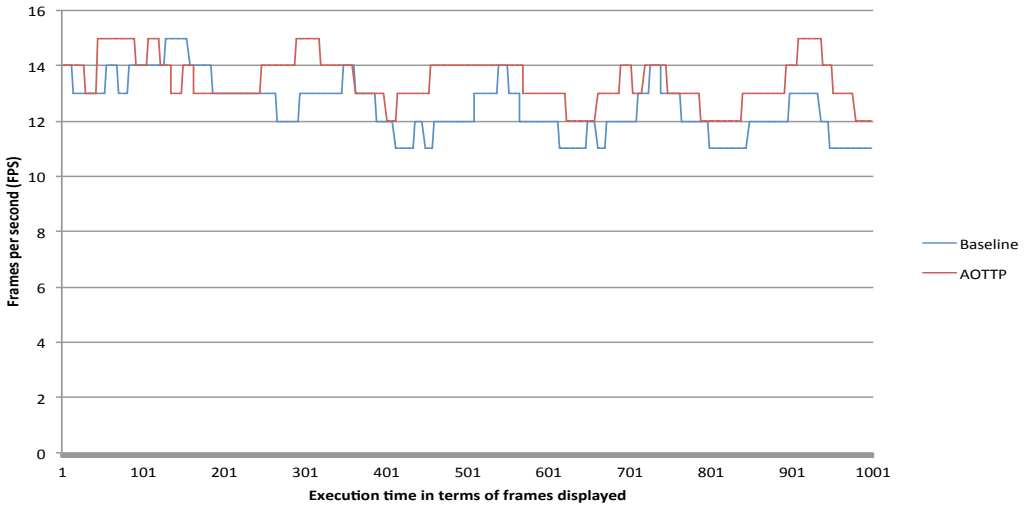

Figure 4.7: Frames per second (FPS) plots for three.js: canvas camera orthographic JavaScript physics engine benchmark. The x axis represents execution times in terms of frames displayed. Higher is better

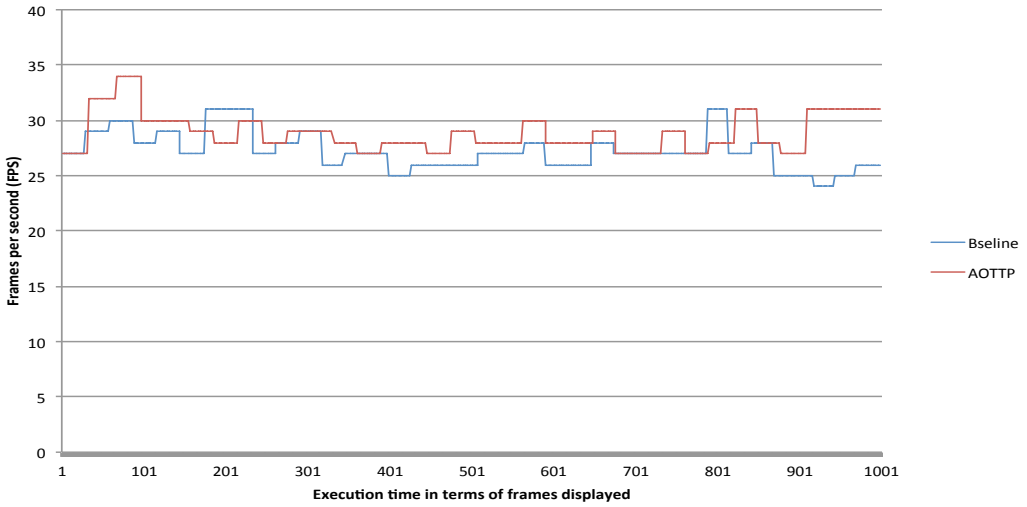

Figure 4.8: Frames per second (FPS) plots for pixi.js: 3D balls JavaScript physics engine benchmarks. The x axis represents execution times in terms of frames displayed. Higher is better

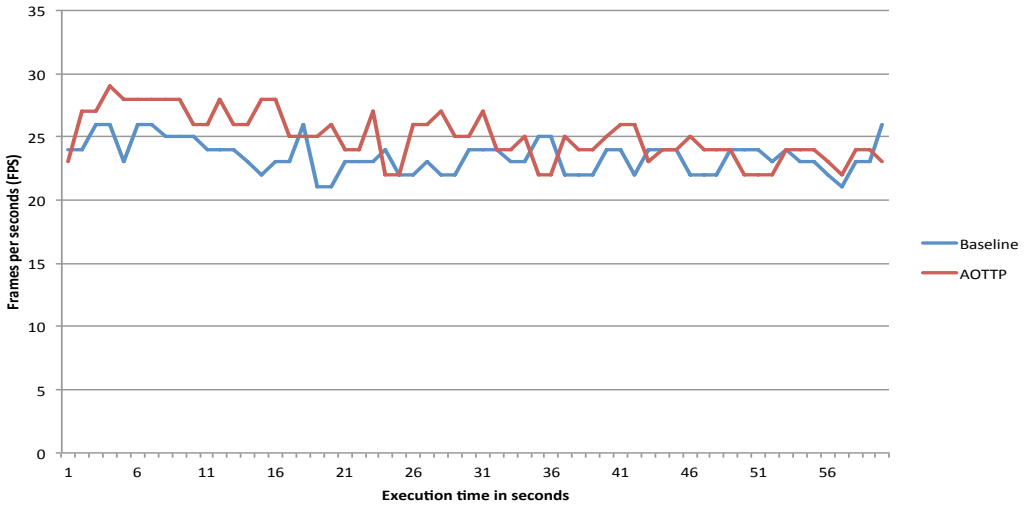

Figure 4.9: Frames per second (FPS) plots for matter.js: multi-body collision JavaScript physics engine benchmarks. The x axis represents execution time in seconds. Higher is better.

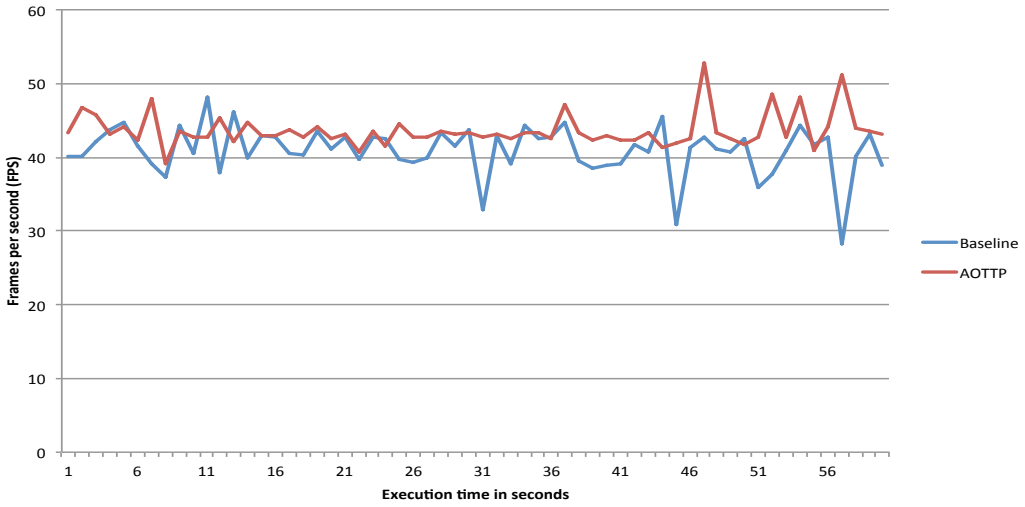

Figure 4.10: Frames per second (FPS) plots for physics.js: multi-body collision JavaScript physics engine benchmarks. The x axis represents execution time in seconds. Higher is better.

Observations Table 4.3 shows the improvement in FPS values for the physics engine demos during the first few seconds of execution. On average, we see 6.75% improvement in the FPS values across all the applications with three.js: canvas ascii and matter.js: multi-body collision benchmarks showing significant improvement in the FPS values during the first 20–30 seconds of execution.

Figures 4.6, 4.7, 4.8, 4.9, and 4.10 show the FPS values seen while executing the physics engine demo applications for the first 60 seconds. The figure shows the FPS values computed for a single representative run. For example, Figure 4.6 shows the evolution of FPS values for the application three.js: canvas ascii. For the first 600 frames the AOTTP configuration shows better FPS values versus the baseline, then gradually the baseline FPS converges to be the same as the AOTTP. This early performance lead by AOTTP shows the effect of optimizing hot functions early. The other benchmarks have similar behavior. This is most easily seen in Figure 4.9; the other benchmarks also converge, but only after several minutes. To keep the graphs legible, we only show the first 60 seconds and so for the other benchmarks the convergence point is not shown.

In some cases the FPS values for the AOTTP optimized version drops below the baseline. This drop is not due to anything inherent in the AOTTP approach, but rather is due to the inherent randomness and dynamism in the benchmarks, such as when exactly garbage collection is triggered. We ran the experiments for each of the benchmarks multiple times and observed similar average speedup in the FPS values for the AOTTP configuration.

Table 4.3 shows the percentage reduction in the deoptimizations for the benchmarks. On average the AOTTP approach avoids 40.82% of the deoptimizations across all the benchmarks. The benchmark pixi.js:3D balls is type stable for most parts and does not have any deoptimizations that can be avoided by AOTTP. But just identifying type-stable hot functions and compiling them eagerly yields a speedup.

Reduction in deoptimizations and the improvement in FPS values do not correlate because of multiple reasons. Some deoptimizations are more critical than others and cause major slowdowns in execution of the program. Avoiding such deoptimizations show significant improvement in performance. Also, in case of three.js:canvas camera orthograhic benchmark, there is no reduction in deoptimizations using our technique. But we see improvement in performance by simply optimizing the hot functions early based on the offline profile information.

As shown in the Table 4.4, the space overhead for the profiles is negligible for all of the benchmarks. The average overhead is only 1.52%.

4.5.5 Membench50

Membench50 is a benchmark suite consisting of 50 real-world popular JavaScript-heavy web pages, primarily designed to evaluate the memory usage of the JavaScript engine. Since our optimization applies to programs that exercise the optimizing compiler, we filter out the websites that have fewer than 30 hot functions.

We use a popular automated website testing tool, Selenium IDE [79] , to simulate user interactions for these websites. For each of the websites, user interactions such as mouse clicks, key presses, and scrolling are recorded using the IDE. These interactions are saved as individual test cases and used as inputs for capturing the offline profile information. Different test cases are used to simulate the user interaction at the client side.

Performing an ideal performance analysis of our approach is difficult in this setting, because we would need to isolate the effects of our optimizations in the presence of network and IO latency in a web browser. Therefore, we present the percentage of deoptimizations that are avoided by our approach as a metric to indicate the effectiveness of our optimization.

Table 4.5 shows the percentage reduction in deoptimizations using the AOTTP approach. In general, most of the benchmarks show reduction in the deoptimization counts while using the AOTTP technique. On average there is a 33.04% reduction in deoptimizations across all benchmarks, with an average profile size of less than 13.08kB. The size of the JavaScript code in these websites are in the order of MB. Therefore, the overhead of profile size is negligible compared to the size of the website. To give a rough estimate, all of the websites present in this benchmark suite use the jquery library. The average profile size is 13% of the size of the jquery library. Therefore, it is safe to assume that the space overhead of the ahead-to-time profile for all of these benchmarks is negligible.

Table 4.5: Experimental results for a subset of Membench50 benchmark suite: number of hot functions, number of deoptimizations in the baseline, number of deoptimizations using AOTTP, percentage reduction in the deoptimizations, and size of the aggregate profile collected using AOTTP approach.

Benchmarks	**# of hot funcs**	**# Baseline deopts**	**# AOTTP deopts**	**Reduction in deopts**	**Profile size (kB)**
businessinsider.com	393	44	20	54.54%	84.05
lufthansa.com	30	12	10	16.66%	3.81
amazon.com	104	23	13	43.47%	21.18
tbpl.mozilla.org	30	17	11	35.29%	2.12
taobao.com	49	18	16	11.11%	4.62
nytimes.com	87	27	24	11.11%	2.27
cnn.com	208	22	18	18.18%	19.76
bild.de	42	2	1	50%	8.49
spiegel.de	32	13	10	23.07%	2.40
lenovo.com	34	0	0	–	3.17
weibo.com	44	4	1	75%	2.37
ask.com	40	4	3	25%	2.82
Average	**91.08**	**15.5**	**10.58**	**33.04%**	**13.08**

4.5.6 Kinds of Deoptimizations

Our technique focuses on type- and shape-based deoptimizations, though other kinds of deoptimizations can happen. We rely on the JavaScript engine to classify each deopti-

mization for us, in order to determine which ones to handle with our profiling technique. One difficulty we encountered specific to SpiderMonkey is that, for technical reasons having to do with the engine's implementation, there are some deoptimizations that the engine leaves unclassified. In other words, for these deoptimizations we do not know whether they were type- or shape-based or some other kind of deoptimization. Our implementation conservatively assumes that they are not type- or shape-based and ignores them. Table 4.6 shows for each Octane benchmark during ahead-of-time profiling the total number of deoptimizations, the percentage that were classified as type- or shape-based, and the percentage that were left unclassified.

We see that a significant portion of the deoptimizations are classified as type- or shape-based, but that an even larger portion are left unclassified. Based on our experience we conjecture that many of these unclassified deoptimizations are actually type- or shape-based and would be amenable to our technique if we could recognize them. However, doing so would require much more extensive changes to the profiling engine (but *not* the client engine). Considering that we get a significant performance benefit just from handling the identified type- and shape-based deoptimizations, it is likely that extending our technique in this way would yield even more significant performance gains.

Table 4.6: Percentage of type- and shape-based deoptimizations (TSDeopt) and percentage of unclassified deoptimizations (UDeopt) verses total number of deoptimizations for Octane benchmark suite.

Benchmarks	**All Deopts**	**TSDeopt**	**UDeopt**
box2d	40	40%	27.5%
crypto	24	33.33%	20.8%
earley-boyer	42	9.5%	9.5%
mandreel	3	0%	100%
typescript	595	9.7%	25.8%
deltablue	4	25%	75%
gbemu	112	17%	11.6%
pdfjs	73	17.8%	17.8%
Average	**111.62**	**19.04%**	**36%**

4.6 Conclusion

We have described a technique to optimize JavaScript programs sent from a server to a client by performing ahead-of-time profiling on the server side in order to reduce deoptimizations on the client side. We have shown that these deoptimizations are an important concern for performance, and that reducing the deoptimizations provides a significant performance benefit. Besides reducing deoptimizations, our technique also allows hot functions to be compiled much earlier than they normally would *and* without having to fear increased deoptimizations due to the reduced profiling time entailed by early compilation.

Our technique relies on several key insights to be practical and effective, such as identifying the correct information to profile to tradeoff the costs and benefits of type profiling and identifying common coding idioms that directly impact the effectiveness of profiling.

We evaluate our technique over three sets of benchmarks: the industry-standard Octane benchmark suite, a set of JavaScript physics engines, and a subset of real-world websites from the Membench50 benchmark suite. Our evaluation shows a maximum speedup of 29% and an average speedup of 13.1% for Octane benchmarks, a maximum improvement of 7.5% and an average improvement of 6.75% in the FPS values for JavaScript physics engines, and an average 33.04% reduction in deoptimizations for the Membench50 benchmarks. The collected profile information is on an average 4% of the size of the JavaScript code.

Chapter 5

Accelerating Server-Side JavaScript

JavaScript is increasingly being used to implement server-side web applications with tailored environments such as `node.js`. However, these applications still run on JavaScript engines that are tuned and optimized for running client-side web scripts, which have very different runtime characteristics than server-side code. In this chapter we investigate techniques with the specific goal of optimizing server-side application performance without requiring heavy modifications to the JavaScript engines they run on.

The `node.js` runtime responds to heavy load by spawning new instances of the JavaScript engine to respond to the incoming requests. Our contribution is a set of techniques for transferring profile information from the original JavaScript engine to the newly spawned engines in order to significantly improve their initial throughput. These techniques provide good performance gains but require only minimal changes to the JavaScript engines.

5.1 Introduction

JavaScript, initially intended solely as a client-side scripting language for web browsers, is now being used to implement server-side web applications that traditionally have been written in languages like Java. `Node.js` [1] is a popular runtime environment for JavaScript server-side applications, originally implemented using Google's V8 JavaScript engine [29]. Other implementations of `node.js` have also arisen, including `Jx-core` [80], which can use either V8 or Mozilla's SpiderMonkey JavaScript engine [14], and `avatar.js` [81], which uses the Nashorn JavaScript engine [13].

A common characteristic of these `node.js` implementations is that they repurpose pre-existing JavaScript engines that were designed for execution in the client-side web browser. These engines are tuned for client-side code characteristics and execution patterns. However, server-side applications do not have the same characteristics and patterns as client-side applications, as demonstrated by Zhu et al [82]. Intuitively, client-side JavaScript engines are tuned to perform well for applications that involve extensive user interaction and DOM manipulation, which do not occur in server-side code. In this paper, we examine techniques for improving the performance of JavaScript engines when being executed on the server, *without* requiring extensive modifications to the engines themselves.

5.1.1 The Problem

To achieve this goal of optimizing server-side application performance, we focus on the response of `node.js` applications when they come under load due to increased traffic. A given `node.js` application is executed in a single thread by the JavaScript engine, though I/O operations are handled asynchronously in different threads. If the number of requests sent to the server exceeds some threshold, the load balancer spawns

new instances of the JavaScript engine, each running an independent instance of the `node.js` application and handling a subset of the requests. These new engines may run on different cores of the same machine or on different machines altogether.

During execution each JavaScript engine employs an online profiler to record types, object shapes, and other information useful for optimizing during JIT compilation. Each newly-spawned engine, then, must take the time to dynamically profile the application in order to optimize it, and this profiling happens every time new engines are spawned due to heavy load. However, the point when a new engine is spawned is exactly when performance is most desirable to maximize the throughput of the new engine to help compensate for the heavy load.

The cost of performing online profiling and waiting for the results to optimize the code hurts the initial throughput. This is a problem faced in real-world applications, as witnessed by numerous blog posts and queries posted on public forums indicating performance problems related to loading new instances of node.js applications [83, 84, 85, 86].

5.1.2 The Opportunity

We observe that all of the engines are running the same application and that the inputs to the application, i.e., the requests, are generally similar to each other. Thus, the profile information collected by the original engine applies equally to all of the other engines—and yet, those engines must still collect this information themselves redundantly, slowing down the application's throughput. The central problem we face in this chapter is how to effectively optimize the newly-spawned engines based on information gathered by the original engine, in order to make sure that the applications have higher initial throughput after being spawned. As an example, consider Figure 5.1. The blue line shows the

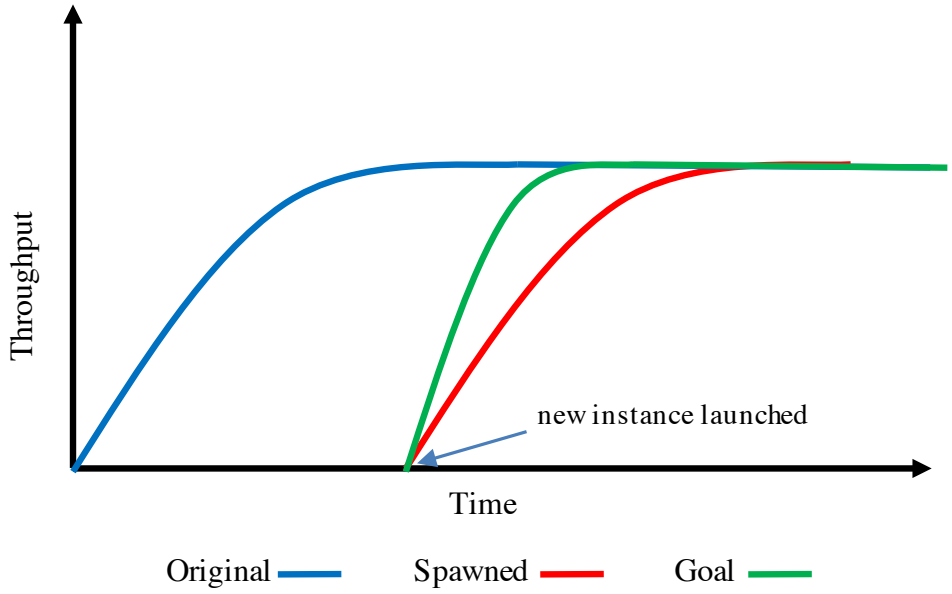

Figure 5.1: Performance of a representative node.js application. The blue line represents the original instance, green line represents our optimized new instance, and the red line represents the unoptimized newly spawned instance.

throughput of the original application instance. There is an initial period where the JavaScript engine is profiling and optimizing the application, then eventually throughput reaches a steady state. The red line shows where a new, independent instance of the application is spawned; it has a similar shape to the original instance. Finally, the green line shows our goal: a newly-spawned instance that immediately attains the optimized throughput of the original instance.

A naïve solution to this problem would be to migrate the already JIT-compiled code from the original engine to the spawned engines (which, again, can be on the same machine or on different machines). However, the compiled code contains concrete references to memory locations in the heap, therefore we would also need to either migrate the heap itself or modify the JIT compiler's code generator. Either way, transferring the compiled code to other engines would require extensive modifications to the JavaScript engine and much engineering effort to make it practical (for example, existing work on

taking snapshots of the heap shows that performance is generally poor compared to not using snapshots [87]).

5.1.3 Our Solution

Our key insight is that we can achieve our goal by sending two specific forms of useful information from the original engine to the spawned engines, which together allow the spawned engines to attain optimized throughput much faster than they could without that information. This information can be transferred simply and cheaply, it results in significant throughput gains during the newly-spawned engines' initial execution times, and it requires only minor modifications to the JavaScript engine implementations.

The first form of information consists of *cachable profile information*, e.g., primitive types, deoptimization information, function hotness, and information about decisions such as method inlining and dense array optimizations. In other words, the obvious information to send that does not require special effort to migrate to a new engine. Cachable profile information is stored in a database and transferred to the new engines as-is; the new engines can read in that information and take advantage of it with only minor tweaks to their implementations.

The second form of information is in lieu of profile information gathered by the original engine that cannot be easily transferred. A prime example is object shape information, which is used for optimizing object accesses in the JIT-compiled code in order to achieve significant performance gains, but that cannot efficiently be transferred to other engines because it is inherently tied to addresses in the heap. Instead, we use a notion of function *type stability*: the point during execution at which a function has enough profile information to be optimized without undue risk of deoptimization. Normally functions may be conservatively executed a large number of times before being

declared hot (e.g., one thousand times by the SpiderMonkey engine); this is to help ensure that functions are type-stable before being optimized. We use information from the original engine's execution to allow the spawned engines to safely optimize functions much earlier than that when possible. In other words, we let the spawned engines know the earliest point in execution at which they have gathered sufficient profile information (of the heap-dependent kind, since we've already sent them the cachable kind) to safely optimize each function. This means that the spawned engine is executing optimized code much sooner than it would otherwise.

5.1.4 Contributions

The specific contributions of this chapter are:

- We describe an overall server-side architecture for JavaScript server applications that allows for increased application performance without extensive engine modification. (Section 5.4)
- We describe a technique for caching profile information and using it across multiple JavaScript engine instances. The profile information includes primitive types, deoptimization information, and boolean (yes-or-no) decisions such as method inlining decisions taken by the optimizing compiler. (Section 5.5)
- We describe two different type stability heuristics for accelerating the compilation of type-stable functions and describe algorithms to determine each of them. (Section 5.6)
- We evaluate our techniques using Jxcore, a SpiderMonkey based `node.js` implementation, and a set of 10 benchmarks that represent a variety of `node.js`

applications. Our evaluation shows significant performance increases for initial throughput for 7 out of 10 benchmarks.(Section 5.7)

Beforehand, we describe some background on JavaScript engine architecture and server-side JavaScript execution (Section 5.2) and some related work (Section 5.3).

5.2 Background

5.2.1 JavaScript Engine Architecture

JavaScript engines rely heavily on online profiling and JIT compilation for performance. Profiling looks at primitive type information for values (`number`, `boolean`, `string`, `object`, `undefined`, and `null`); object shape information (what object properties are present and their offsets); and more. JavaScript engines usually employ multi-tier architecture for execution, as described in Figure 5.2.

Tier 1. The first tier of execution is a fast interpreter for a parser-generated intermediate representation. The goal of an interpreter is to execute functions quickly so that the user does not have to wait until the function is compiled. This strategy is useful for client-side JavaScript code, but not for server-side JavaScript code where the main emphasis is on the peak performance of. In fact, the V8 JavaScript engine, which is used in the official `node.js` implementation, does not employ this tier of execution.

Tier 2. A baseline compiler and baseline code execution forms the second tier of execution. In this tier, the functions are compiled to generate chains of stub code by the baseline compiler. During the baseline execution the stubs are replaced by machine code that represent type specialized operations on the fly as they are executed. The baseline code also acts as a profiler that collects type information and stores it in a form that is

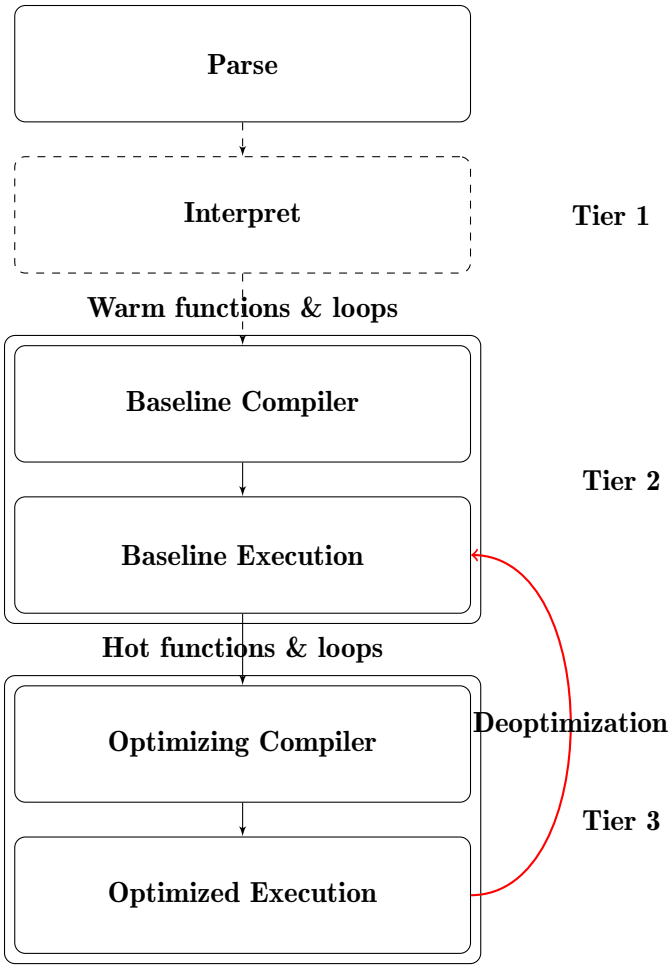

Figure 5.2: Flow graph showing different phases of execution in a generic JavaScript engine. The interpretation phase, represented by dashed lines, is an optional phase in JavaScript engines like Google's V8.

easily accessible to the optimizing compiler. This type information includes the types of individual variables that are used in the function and the shapes of objects present at different program points in the function. In addition to collecting profile information, the baseline code also acts as an entry point for execution to resume after deoptimization.

Tier 3. This tier consists of an optimizing compiler that uses the profile information collected in tier 2 to generate type-specialized highly optimized code. This code executes orders of magnitude faster then the naive interpreted code. The type profile information

collected at tier 2 is used to perform optimizations such as type specific arithmetic operations, reducing boxing and unboxing of dynamic values, object property access inlining, and more. Because these optimizations rely on profiled information, it is possible during the execution the runtime encounters types that were not observed before. This causes the compiled code to be invalid and the runtime bails out of the specialized code, jumping to the corresponding point in the baseline code and contuining execution. This operation is called deoptimization.

5.2.2 Server-side JavaScript Execution

The original `node.js` is a server-side programming platform built on top of the V8 JavaScript engine. `Node.js` adds essential features such as APIs for disk access, networking, and inter-process communication; these allow the programmer to use JavaScript to build server applications as well as client code. These APIs are non-blocking, therefore the programmer must specify callback functions to handle the return values of these APIs when invoked. This unique asynchronous model of execution is also known as the `node.js` programming paradigm. There are various implementations of this paradigm that use different JavaScript engines. The original `node.js` uses Google's V8 JavaScript engine, `Jxcore` can use either Mozilla's SpiderMonkey or the V8 JavaScript engine, and OperJDK's `Nashorn` JavaScript engine has it's own implementation of `node.js` using `avatar.js`. For the rest of this chapter, `node.js` refers to the programming paradigm and not the specific implementation of it.

Figure 5.3 shows a typical execution pattern for a `node.js` application. Given the asynchronous nature of the application, `node.js` gives an illusion of multithreaded execution to the programmer. In reality, for each instance of `node.js`, only one thread performs JavaScript execution. The `node.js` runtime launches auxiliary threads to

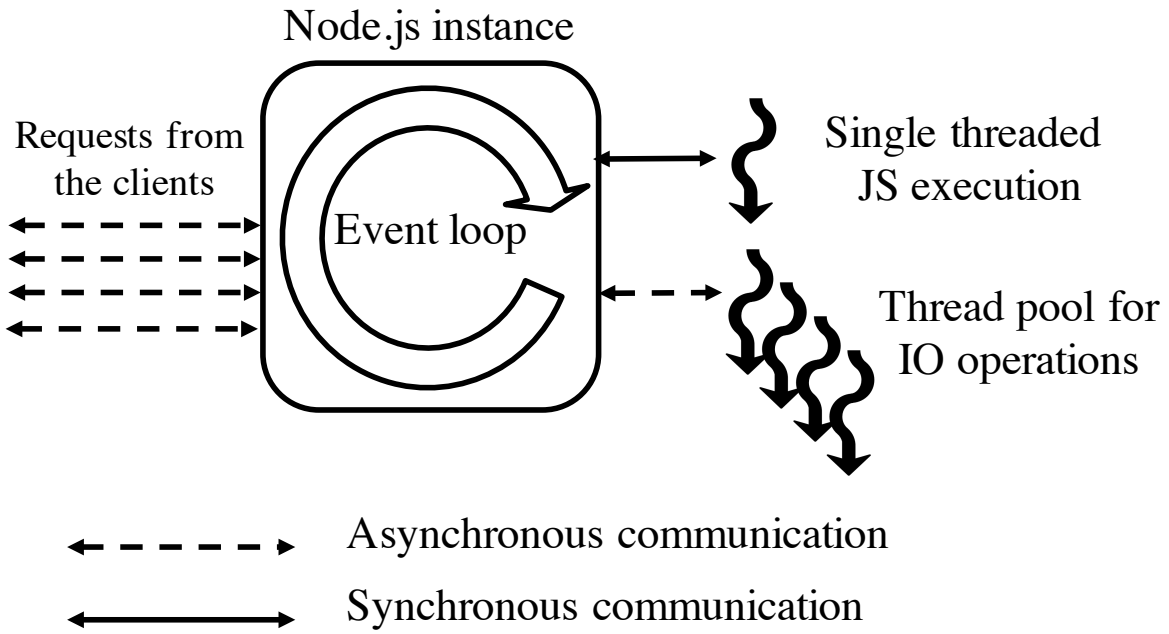

Figure 5.3: `node.js` execution model in which JavaScript esecution happens in a single thread and IO operations are carried out in separate threads from a thread pool.

handle non-blocking IO operations. Each function in the server-side application code executes in the JavaScript engine and goes through the various phases of execution as described in Section 5.2.1.

Figure 5.4 shows the `node.js` execution pattern for large applications that require load balancing. Typically, when too many requests come in and exceed some threshold, the load-balancer [88, 89, 90, 91] automatically launches new instances of the application running in newly-spawned JavaScript engine instances, possibly on entirely different machines. When new instances are launched, the JavaScript functions in the server code go through the various phases of execution in the underlying JavaScript engine all over again. Therefore, the JavaScript engine has to collect fresh profile information for each of the instances that are launched in order to generate optimized code.

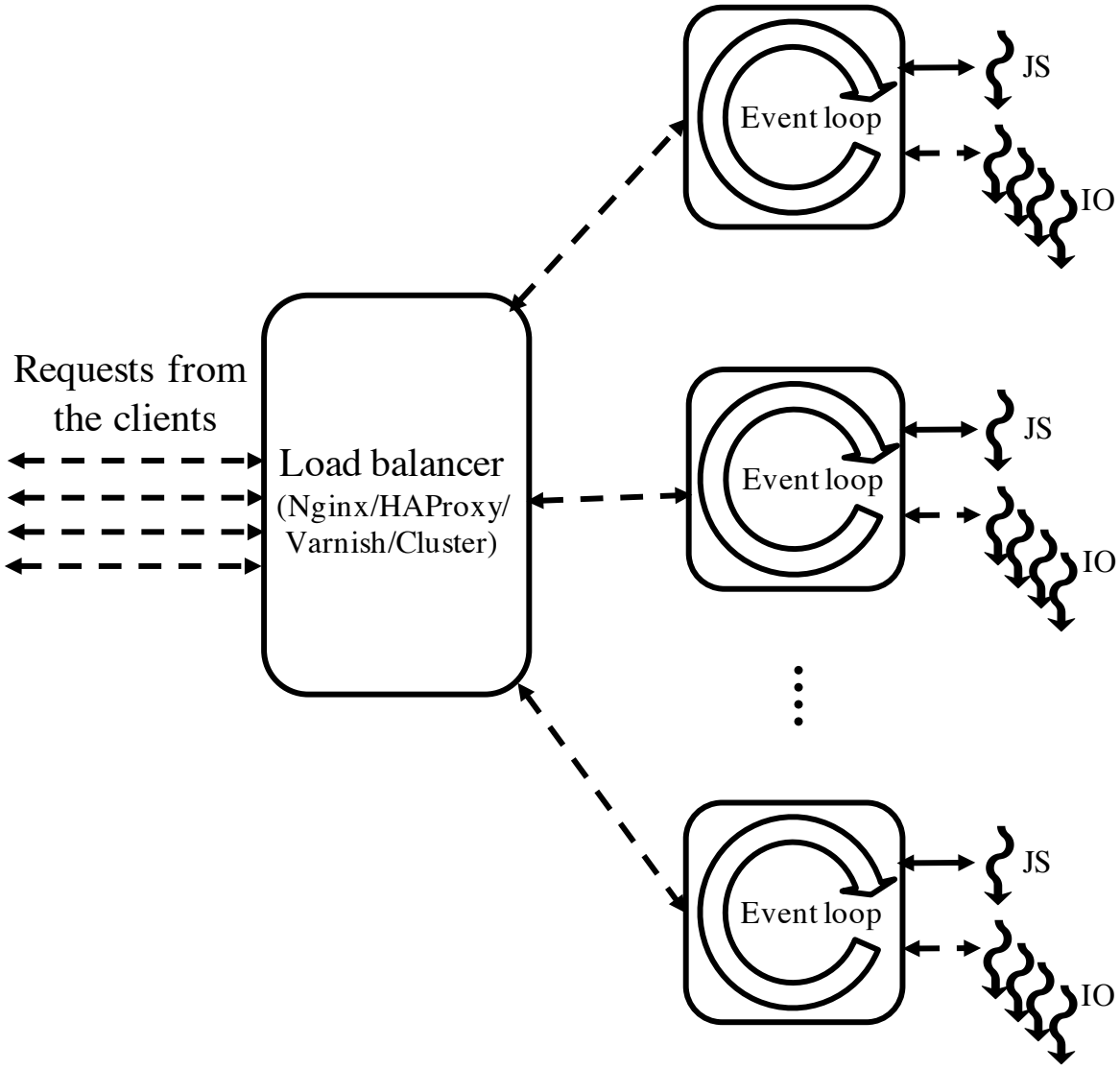

Figure 5.4: Load balancer launches new instances of `node.js` depending on the number of requests that are sent to the server. Each instance runs on a single processor and has a JavaScript engine executing in a single thread. Note that depending on the configuration, the new instances may be launched on different machines.

5.3 Related Work

In this section we describe work on optimizing JavaScript and Java that is related to either our ultimate goal of optimizing server-side applications or that explore similar strategies as those we employ in our own work.

5.3.1 JavaScript

In the previous chapter, we experimented with server-side profiling of client-side JavaScript for client-side web applications. Unlike this project, server-side profiling did not examine the problem of server-side applications. In the previous chapter, we use a notion of type stability, but our technique for determining type stability requires an ahead-of-time profiling phase. There is also an additional stability testing and analysis phase to figure out the minimal profile information that is required to be annotated in the source code. The technique described in this chapter is online, therefore it is required to be non-intrusive and fast, and it specifically targets server-side applications and the server setting.

Snapshot-based code migration is an alternative approach to code migration. Oh et al [69] describe a snapshot-based code caching mechanism which significantly accelerates the loading time of client-side JavaScript applications for various JavaScript frameworks. It is not clear whether the snapshot-based approach is feasible in the presence of a multi-layered JIT architecture as usually present in server-side JavaScript engines. One other drawback of this approach is that the process of taking snapshots is an expensive and tedious process. Oh et al report a 29–44% overhead in execution time for framework code.

Developers have experimented with a snapshot-based approach for migrating `node.js` applications using the `nwjc` tool [87]. However, the developers report a slowdown of

aound 30% for various benchmarks. Also, the developers report incompatibility of snapshots across different versions of `node.js` and platforms.

5.3.2 Java

Arnold et al [57] describe a cross-run repository of profile information for the Java Virtual Machine. The main idea of the paper is to record and retain raw profile data such as time spent in various methods of the program, call targets at virtual call sites etc. During the VM shutdown phase, the recorded information is analyzed and a new aggregate *pre-computed online strategies repository* is created. This repository is used by the instances of the applications that are launched later to make clever decisions while analyzing and compiling hot functions. Unlike this approach, our strategy does not have a separate analysis phase. The profile information is recorded as and when the program is executed. The repository is occasionally pruned to remove unnecessary profile data using standard database filtering techniques. Therefore, the profile information is always usable and the new instances do not have to wait fo the monitoring server to shutdown in order to obtain the analyzed data. Furthermore, unlike their technique, our approach is specifically designed to handle long running server-side applications written in a dynamically-typed language.

Stephenson et al [92] describe a machine learning based approach to determining compilation heuristics for the Java virtual machine. In this approach, machine learning is used to determine if and when compiler optimizations are applicable to certain class of applications using data from various applications as a training set. Machine learning is a complicated approach and requires a lot of resources and time. Therefore, our approach is faster, simpler, and more conducive for server-side JavaScript applications that need to be launched instantaneously.

Sandya et al [93] describe a method of combining offline and online profile information to guide the compilation heuristics of a Java virtual machine. The paper specifically deals with minimizing the impact of compilation time for potential hot functions on the overall execution of the application. Modern day JavaScript engines employ concurrent compilation and, unlike type stability, the compilation time is not a major concern. The paper also talks about modifying the heuristics to use temporal event profiles to handle phase changes in the application. Though we do not deal with phase changes in this paper, this can be viewed as a possible extension to our work.

5.4 Our Technique's Overall Architecture

Figure 5.5 shows a diagram describing the overall architecture of our approach. An initial JavaScript engine instance runs the server-side application; this instance is modified to record its online profile information into a profile database (the specifics of this information are discussed in Sections 5.5 and 5.6). If the load balancer launches a new JavaScript engine instance on the same machine as the monitoring instance, the new instance reads the profile information from the profile database as needed. If the load balancer launches a new instance on a different machine, then the profile database is copied over to that new machine as well and the new instance reads the profile information from that copy as needed.

5.4.1 Running Example

We will use a running example throughout the next two sections that will be used to explain the details of our approach. Figure 5.6 shows a dummy JavaScript program that

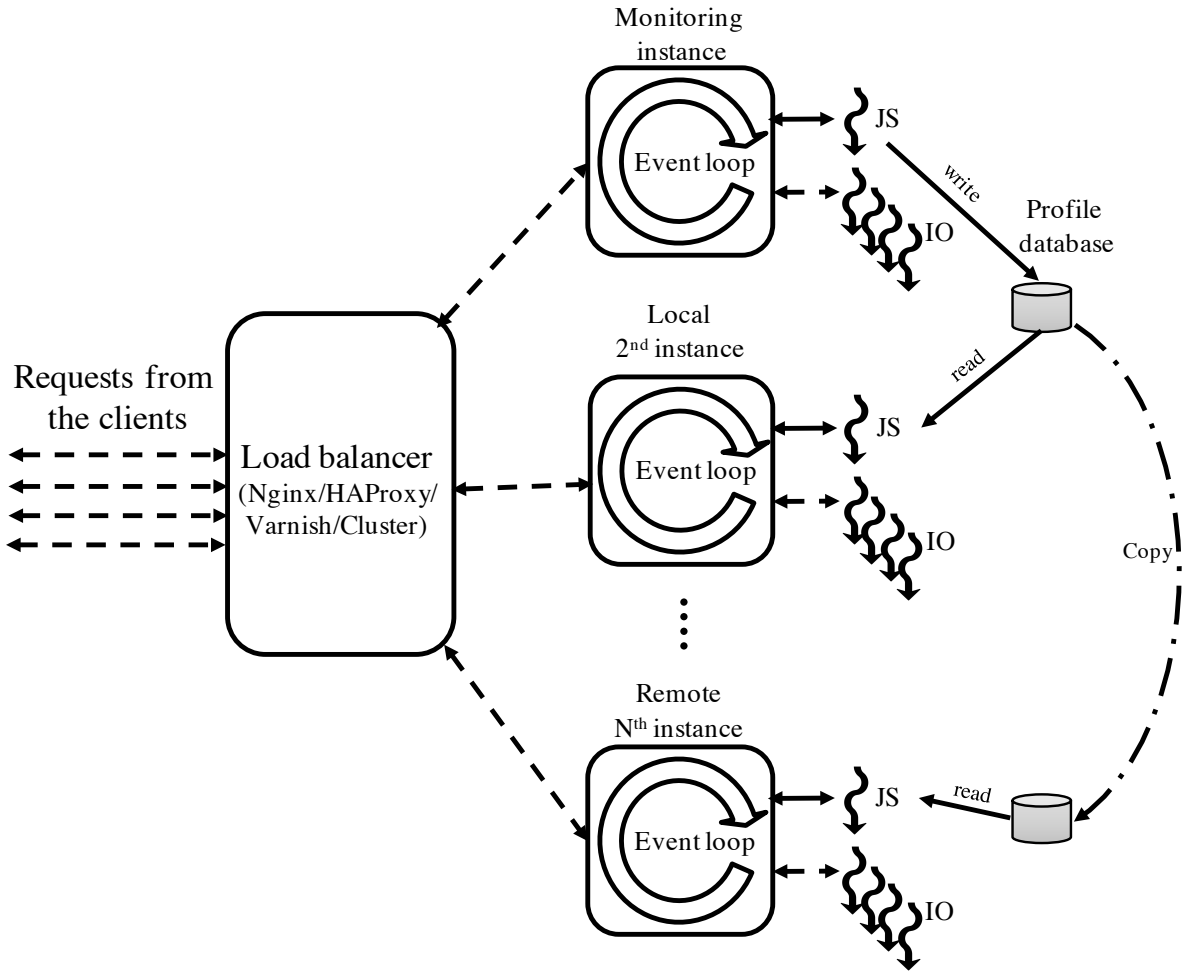

Figure 5.5: During the first monitoring run the profiling database is populated. When new instances are launched by the load balancer, if the instance is launched in a new machine, the database is copied over and the new instance is configured to access the database. Otherwise, the new instance is configured to access the original profiling database.

serves as a stand-in for some server-side application.[1] This dummy code is executed by the original JavaScript engine instance as well as any engine instances spawned by the load balancer.

Line 1 shows a JavaScript function `hotFun` that is called 2,000 times in a loop in lines 12, 14, and 16. During the first 1,000 times, `hotFun` is profiled to collect information about primitive types, object shapes, etc. After the 1,000th time, `hotFun` is deemed hot and is compiled by the optimizing compiler using the collected profile information. This threshold of 1,000 is taken from the Mozilla SpiderMonkey JavaScript engine, which is representative of mainline JavaScript engine implementations.

The loop at line 10 is also hot and would also be compiled by the optimizing compiler. However, for the sake of simplicity we assume that the loop represents the `node.js` event loop and we do not consider its compilation or optimization.

5.5 Cachable Profiling Information

Cachable profiling information consists of data that is gathered by the JavaScript engine's online profiler that is not dependent on the current state of the heap or any data structure that resides in memory. Thus, this information can easily be stored in an external database and used by other JavaScript engines. In this section we describe which information gathered by the online profiler can be considered cachable and how it is used.

Primitive types. Primitive types as recorded by the profiler are used to type-specialize code generated by the optimizing compiler. A simple example is the addition operator. If the profiler records at a certain program point that the types of the two operands for

[1] The program is not actual server-side code, which is event-driven and would unnecessarily complicate the presentation of our techniques.

```
(1)     function hotFun(a, obj) {
(2)         var inc = a + 1;
(3)         inc = obj.x + bar();
(4)         return inc;
        }

(5)     function bar() {
(6)         return SOME_INT_VALUE;
        }

(7)     var point = {x:0, y:0};
(8)     var zpoint = {x:0, y:0, z:0};
(9)     var ind = 0;

(10)    for (;ind < 2000; ++ind){
(11)        if (ind < 100)
(12)            hotFunc(ind, point)
(13)        else if (ind < 500)
(14)            hotFunc(ind, zpoint);
(15)        else
(16)            hotFunc(2.5, zpoint)
        }
```

Figure 5.6: Functions `hotFun` and `bar` are invoked 2,000 times. `hotFun` becomes hot after 1,000 iterations. The online profiler captures profile information at lines 2 and 3 for the function `hotFun` which is used by the optimizing compiler to optimize it.

that operator have always been integers, the optimizing compiler can use this information to generate type-specialized integer arithmetic code. Because primitive types like `int`, `bool`, `double`, `null`, and `undefined` can be represented using simple enum-like constructs, it is easy to capture them in the profiling database. The information that is captured in the database is of the format `<filename, lineNo, columnNo, pcOffset, type>`, where the first three fields designate a unique function, the fourth field designates a point within that function, and the last field designates the enum value representing a primitive type.

In the running example from Section 5.4.1, for the first 500 iterations the profiler records `a` to be `int` at line number 2, `obj.x` and the return value of `bar` to be `int` at line 3, and the resultant type of the addition at lines 2 and 3 to be `int`s. For the remaining iterations, the profiler records that `a` is a `double`.

Deoptimization information. The profiler records function deoptimization information in order to keep track of how effective the engine's optimization decisions were. Deoptimization happens when an assumption made by the optimizing compiler (based on information provided by the profiler) turns out to be incorrect, rendering the optimized code invalid and requiring the engine to revert back to the unoptimized code. This can happen for a number of reasons. The ones we focus on in this chapter are:

1. A new primitive type could be observed in future invocations of the function that wasn't seen in previous invocations.

2. An overflow could occur after an arithmetic operation which was assumed to result in an integer but now requires a floating point value.

3. Array bounds could have been violated after having the bounds checks optimized out.

4. Fast object property access could fail to account for new object shapes that are seen in future invocations.

Information about the above problems can potentially enable the optimizing compiler to make better decisions while generating the optimized code:

1. If a new primitive type is observed after type specialization, the profile database records the new type along with the program point where it occurred. The optimizer for the newly spawned engine can then account for that type when the function is compiled. The program point is represented as a tuple `<filename, lineNo, columnNo, pcOffset>`, where the first three fields designate a unique function and the last field designates a point within that function. The profile database, then, contains a mapping from program points to sets of observed primitive types that caused deoptimization.

2. If an arithmetic operation causes integer overflow and thus deoptimization then the profile database records the program point where that happened along with a flag indicating the integer overflow problem. The optimizer for the newly spawned engines can then take care to use floating point operations instead of integer operations, thus avoiding the deoptimization.

3. If an array access at a particular program point causes deoptimization then the profile database records that fact and the newly spawned engine uses it similarly to integer overflow as described above.

4. If an unanticipated object shape causes deoptimization (because the object does not have the proper layout assumed by the optimizer) then again the program point is recorded in the profile database and the newly spawned engine's optimizer uses

this information to avoid using the fast object property access code that assumes a specific layout, thus avoiding the deoptimization.

In the running example, if we assume that `bar` returns a large `int` value that caused an integer overflow during the execution of function `hotFun` during the monitoring run, the profile database can be updated with this information so that in the subsequent runs the optimizing compiler can generate code to avoid deoptimizations due to overflow.

Inlining information. The JavaScript engine's optimizing compiler uses heuristics to decide whether to inline callee functions at certain program points. These heuristics are based on size and hotness of the callee function. At the time the caller function is optimized, if the callee function is not marked as hot and it is considered too large then it will not be inlined into the caller function.

However, it may happen that the callee function becomes hot *after* the caller function is optimized. This is a missed opportunity for optimization. The profile database records instances where a callee function at a certain program point was *not* inlined and yet later became hot; the newly spawned engine's optimizer can use this information to make a better inlining decision for itself.

5.6 Type Stability

There is important information collected by the engine's online profiler that is inherently dependent on the heap, and thus cannot be simply transferred from one engine to another. Object shapes (also known as hidden classes) and object type representations are examples of such information. Because we cannot transfer this information from the original engine instance to the newly spawned engines, we must allow each newly spawned engine to rediscover this information for themselves.

However, we *are* able to optimize the way in which the newly spawned engines collect and make use of this information based on knowledge gained from the original engine. Specifically, we can influence the time that it takes for the spawned engines to optimize hot functions. Naïvely, we could simply mark hot functions in the original engine and have the spawned engines compile those hot functions with the optimizing compiler immediately upon being spawned. The problem is that without all of the profile information available to the original engine (including information that cannot be transferred via the profile database) a spawned engine's optimizing compiler cannot make good decisions.

Recall that during normal operation, a function (or loop) must execute for one thousand times before being considered hot and compiled with the optimizing compiler. This duration is set to help ensure that all of the necessary profile information has been collected *before* the function is optimized. In other words, the engine is waiting until the function is *type stable*. In many cases, however, this duration is too conservative—the appropriate information has been collected well before that threshold is reached.

This gap presents an opportunity: we can use knowledge from the original engine's execution to have the spawned engines safely optimize hot functions without having to wait for the full one thousand invocations. To accomplish this, we can send information to the spawned engines indicating a specific criteria for marking a particular function as type stable and hence ready to be optimized, where that criteria is based on the original engine's experience with that function.

We have experimented with two different forms of type stability criteria which we detail below. These two proposed criteria attempt to find a sweet spot between being easy to communicate and track and being precise enough to be useful. Our evaluation in Section 5.7 experiments with both forms of criteria.

5.6.1 Invocation Count Type Stability (ICTS)

The first criterion we can use to determine type stability is *invocation count*. For each hot function, the original engine determines the first function invocation at which the function became type stable before being optimized and communicates that value to the spawned engines. When that function reaches the given invocation count in the spawned engine, it is optimized without waiting for the one thousand invocation threshold. This criterion is easy to track (being a simple counter per function, which is already implemented by the JavaScript engine in order to determine hotness) and to communicate to spawned engines (in the form of a mapping from function to integer).

In order to determine the correct invocation count, the original engine constantly updates the type stability value for a function with the last invocation count that caused heap-dependent profile information to be recorded by the online profiler. Algorithm 3 shows how the original engine computes the invocation count for a function. The function `UpdateTypeStabInvocCount` is called whenever new heap-dependent profile information is recorded by the online profiler. Algorithm 4 shows how spawned engines use the count to determine when to compile a function. The function `IsTypeStable` is called before every invocation of the function in the baseline code.

Algorithm 3 ICTS algorithm—Original Engine (F is the function, PD is the profile database)

```
procedure UpdateTypeStabInvocCount(F, PD)
    invocationCount ← PD(F).invocationCount
    if F.invocationCount ≥ invocationCount then
        PD(F).invocationCount ← invocationCount
    end if
end procedure
```

In the running example from Figure 2.2, the function `bar` becomes type stable after the first iteration, whereas the function `hotFun` becomes type stable after 101 iterations. The type of the parameter `a` changes to a `double` after 500 iterations. But this type

Algorithm 4 ICTS algorithm—Spawned Engine (F is the function and PD is the profile database)

```
procedure IsTypeStable(F, PD)
    invocationCount ← PD(F).invocationCount
    if F.invocationCount ≥ invocationCount then
        return true
    else return false
    end if
end procedure
```

change is not considered while calculating the type stability metric because the double type is cachable information and is captured in the profile database.

5.6.2 Type Profile Count Type Stability (TPCTS)

The alternate criterion we can use to determine type stability is *type profile count*. This is a finer-grained metric than invocation count; it is more complex and requires more work to track and communicate, but it is more precise than the first criterion.

For each function, for each program point, the original engine counts how many different instances of heap-dependent profile information are tracked at that program point before the function becomes type stable and is optimized. These counts are stored in the profile database as a mapping from program point to integer and sent to the spawned engines. The spawned engines keep a counter per program point that is incremented whenever the online profiler records new heap-dependent information. When the counters for every program point of a function equal or exceed the value in the profile database, the function is marked hot and optimized. Algorithm 5 shows how the original engine computes the type profile count for a program point of a function. The function `UpdateTypeProfileCount` is called by the online profiler whenever *new* heap-dependent profile information is recorded at a program point. Algorithm 6 shows how spawned engines use the count to determine when to compile a function. The function

`IsTypeStable` is called before every invocation of the function in the baseline code. If the function is already classified as hot by the engine's online profiler, the algorithm skips the type stability check.

Algorithm 5 TPCTS algorithm—Original Engine (F is the function, PD is the profile database, and PC is the program counter)

```
procedure UpdateTypeProfileCount(F,PD,PC)
    PD(F, PC).profCount ← PD(F, PC).profCount + 1
end procedure
```

Algorithm 6 TPCTS algorithm—Spawned Engine (F is the function and PD is the profile database)

```
procedure IsTypeStable(F, PD)
    if IsFunctionHot(F) then
        return true
    end if

    for all pc ∈F.nonCachableProfilePCs do
        if F(pc).profCount < PD(F, pc).profCount then
            return false
        end if
    end for
    return true
end procedure
```

Because the spawned engines are not executing exactly the same requests in the same order as the original engine, the provided type profile counts are not guaranteed to be completely accurate. However, this criterion is more precise than the invocation count criterion and will result in fewer deoptimizations due to premature optimization.

In the running example, during the original engine's execution two shapes are recorded for `obj` (one corresponding to `point` and another corresponding to `zpoint` object) and one closure is recorded for `bar` in line 3. During the spawned engines' executions, at the moment when two shapes have been recorded for `obj` and one closure has been recorded for `bar` by the online profiler then the function is marked as type stable and compiled

by the optimizing compiler. In our example, both ICTS and TPCTS mark `hotFun` to be type stable after 101 iterations.

5.7 Evaluation

We evaluate our implementation on `Jxcore` [80], an implementation of `node.js` which uses Mozilla's SpiderMonkey [14] JavaScript engine. We use the standard `node.js` performance benchmarks from the `Jxcore` official repository.[2] From those we select the 10 benchmarks that allow the execution duration to be configured, in order to properly simulate the relation between the original JavaScript engine execution and the newly spawned engines (i.e., the original engine will have executed longer than the newly-spawned engines).

Experimental Setup. We run our experiments on an 8-core Intel i7-4790 machine with 32GB RAM running Ubuntu 14.04 Linux operating system. For the original run, the benchmark is executed with training configuration and inputs for 20 seconds. The profile database that is captured is used for subsequent newly-spawned instances as it closely resembles the actual execution pattern of a long running `node.js` application.

Calculating Throughput. To simulate the newly-spawned engines, each of the benchmarks is executed using a different configuration and inputs from the execution of the original engine, with access to the profile database computed by the original engine as described above. Each benchmark is executed 10 times, and we record the average throughput at regular intervals during the execution.

Configurations. We run our experiments with the following five configurations:

[2] https://github.com/jxcore/jxcore/tree/master/benchmark

- **Original:** the unmodified application running on a vanilla `Jxcore` implementation.
- **Original with ICTS profiling:** representing the modified original engine with ICTS profiling enabled.
- **Newly-spawned with ICTS:** representing the newly-spawned engine using the ICTS profile information.
- **Original with TPCTS profiling:** representing the modified original engine with TPCTS profiling enabled.
- **Newly-spawned with TPCTS:** representing the newly-spawned engine using the TPCTS profile information.

5.7.1 Throughput Improvements

Figures 5.7.1 and 5.7.1 show the throughput for various configurations of the benchmarks during the first three seconds of their execution. For most of the benchmarks the newly-spawned ICTS and TPCTS engines perform very well compared to the baseline. The throughput usually varies during the execution of the application and is influenced by various internal and external factors like disk latency and garbage collection. Therefore, the throughput values can vary over time for different configurations.

A common trend among most of the benchmarks is the initial spike in the throughput value after 250ms of execution. We believe this spike is because during the initial stages of execution of the benchmark, the garbage collector (GC) is dormant and doesn't interfere with the execution of the application.

Outliers: One definite outlier among the benchmarks is the `client-request-body.js` benchmark in Figure 5.13. For this benchmark, all the configurations show similar be-

havior during all the phases of execution. This is mostly because this application only has one hot function. Therefore, there is not much overhead while collecting the profile information and there isn't much speedup obtained by using the profile information to optimize just one hot function in the newly-spawned configurations.

The newly-spawned ICTS and TPCTS instances show average to poor performance for the `net-s2c.js` and `net-pipe.js` benchmarks. Figures 5.10 and 5.11 show a sudden dip in throughput after the initial spike at 250ms. On further investigation we realized that the GC was at fault for the poor performance. For these benchmarks, the GC always triggers as soon as hot functions are compiled by the optimizing compiler (even for the vanilla `Jxcore` runtime, without any of our modifications). In SpiderMonkey compiled code is garbage collected at every major collection, and thus for these two benchmarks, for all configurations, the highly optimized code is garbage collected as soon as it is produced.

This is a problem for any JavaScript engine running these benchmarks even without our modifications, but our modifications exacerbate the problem precisely because we cause functions to be compiled more quickly, thus causing GC to happen more frequently and causing the optimized code to be thrown away again in a vicious cycle.

Observations: Intuitively, we expect that TPCTS should perform better than ICTS because the profile information provided to the TPCTS instance is more precise. However, for the benchmarks `tls-throughput.js`, `dgram.js`, and `net-c2s.js` the newly-spawned TPCTS instances have lower performance compared to the newly-spawned ICTS engines. The following is the reason why this happens. During TPCTS profiling in the original instance, all the heap-dependent information collected at a program point is counted without any consideration for when the information is recorded during the execution of the program. This may lead to a problem if a function was monomorphic during

the initial phases of execution of the application and becomes polymorphic during the later phases. In the newly-spawned instance, the function is not regarded as type stable if all the heap-dependent information is not recorded at every program point. Therefore, in the initial phases the function is not regarded as type stable leading to poor performance.

For all the benchmarks, the performance of the original instance with ICTS profiling is comparable to the original instance. This is because we use fast in-memory database to capture the profile information and backup the database occasionally in a separate thread to minimize the impact of disk latency during the execution of the program. But the original instance with TPCTS profiling enabled performs slightly worse. This is expected because more information is recorded at every program point for TPCTS when compared to ICTS.

Newly-spawned TPCTS engines perform extremely well for the `cipher-stream.js` benchmark. Upon inspection we found that the benchmark is mostly static and monomorphic in nature and this nature of the benchmark does not change with respect to the inputs provided, which is a best-case scenario for our technique.

5.7.2 Function Compilation & Type Stability

To measure how soon functions are compiled, we modified the original engine and the newly-spawned engines using the ICTS and TPCTS heuristics to record the invocation count when the function is compiled by the optimizing compiler. A few functions that require no profile information to execute are statically analyzed by the engine and compiled after just one invocation.

Figure 5.17 shows the how soon the functions are compiled for the original and newly-spawned ICTS and TPCTS engines for all of the benchmarks combined. In the original engine, most of the functions are compiled after one thousand invocations. For the newly-

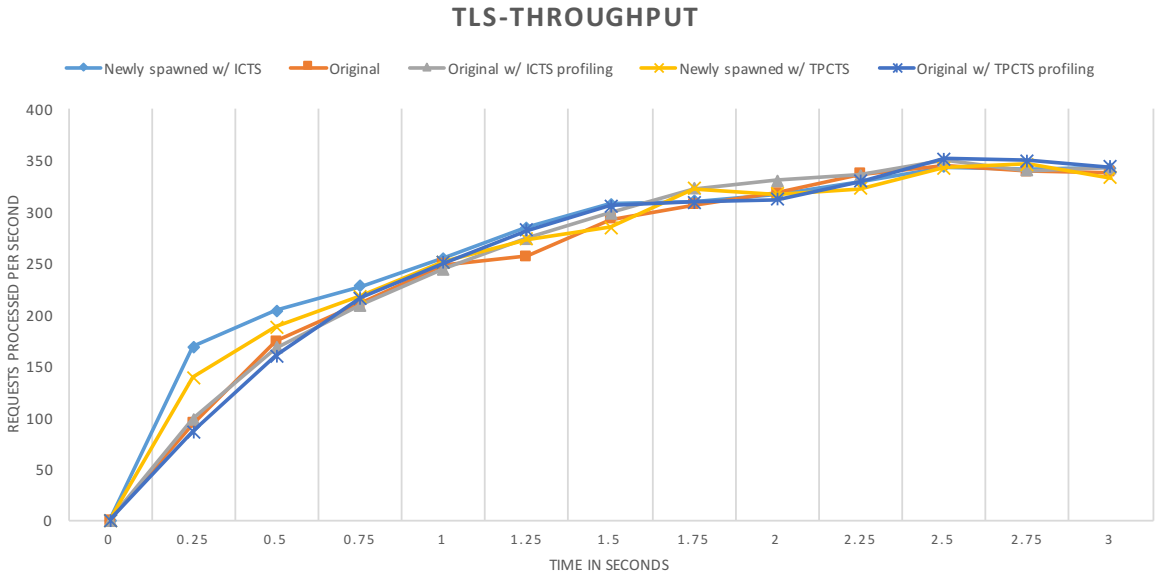

Figure 5.7: Operations per second plots for tls-throughput.js. The y axis represents the requests processed per second values for a TLS server.The x axis represents execution time in seconds. Higher is better.

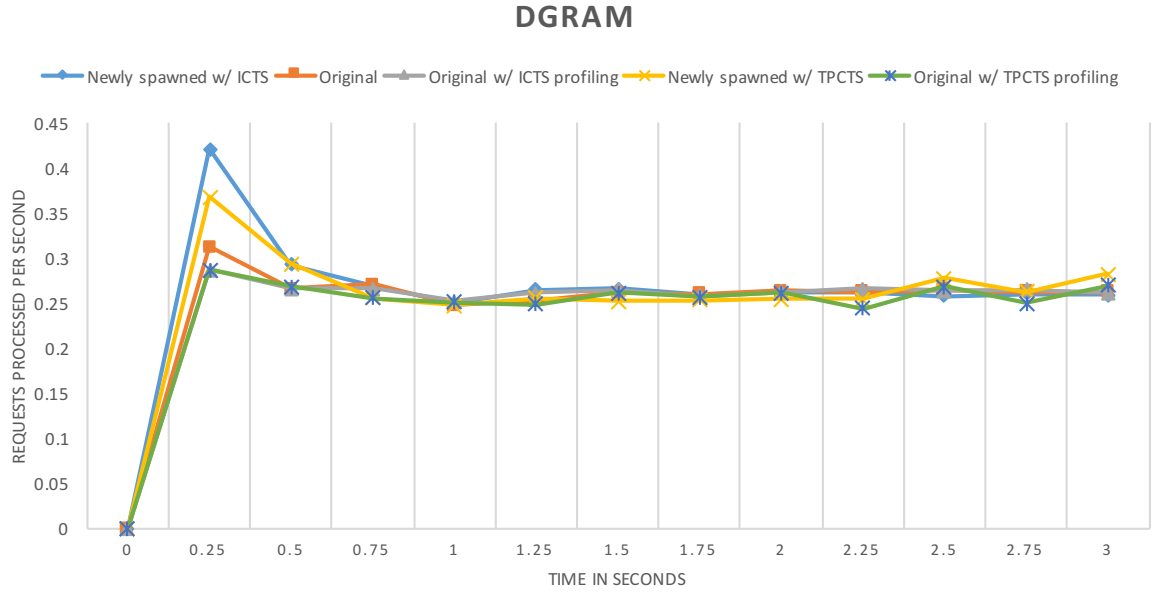

Figure 5.8: Operations per second plots for dgram.js benchmark. The y axis for 5.8 represents the requests processed per second values for a UPD server. The x axis represents execution time in seconds. Higher is better.

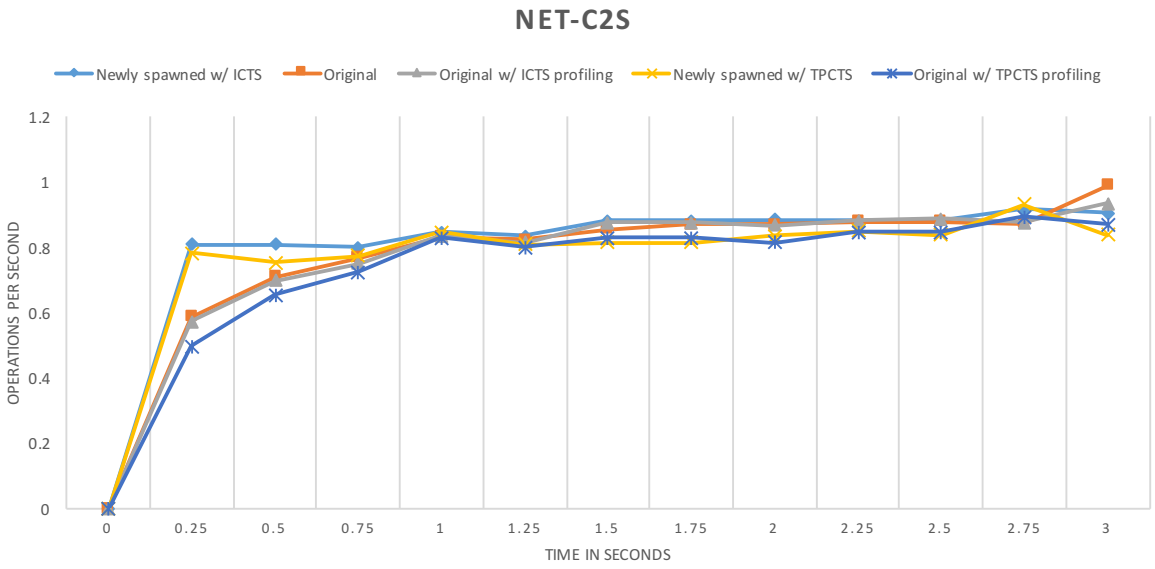

Figure 5.9: Operations per second plots for net-c2s.js benchmark. The y axis represents the number of API operations performed by the benchmark per second. The x axis represents execution time in seconds. Higher is better.

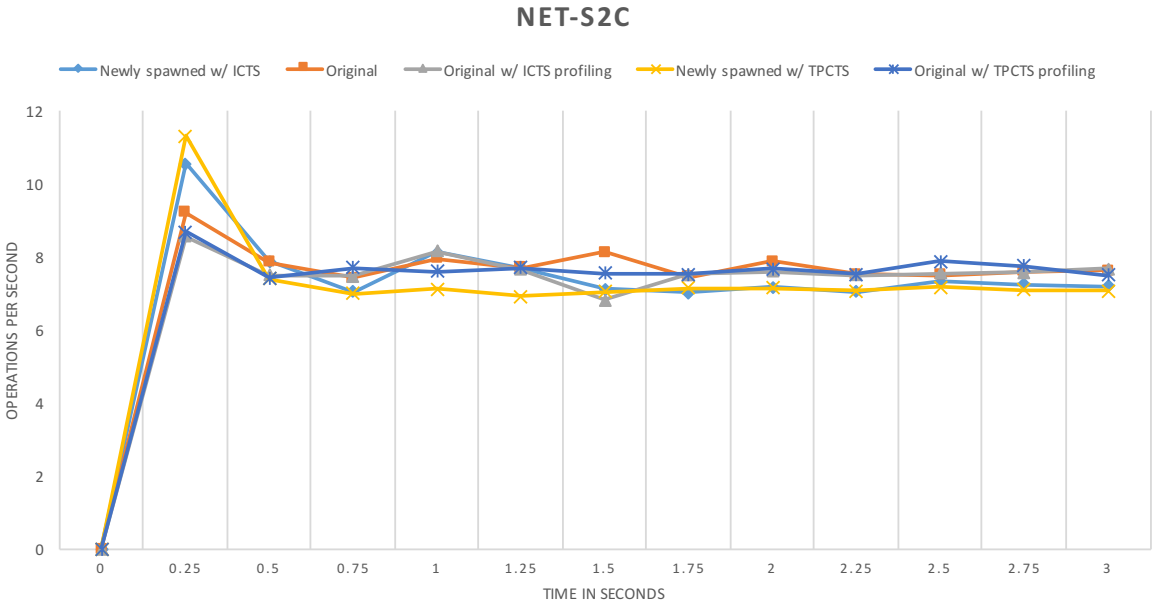

Figure 5.10: Operations per second plots for net-s2c.js benchmark. The y axis represents the number of API operations performed by the benchmark per second. The x axis represents execution time in seconds. Higher is better.

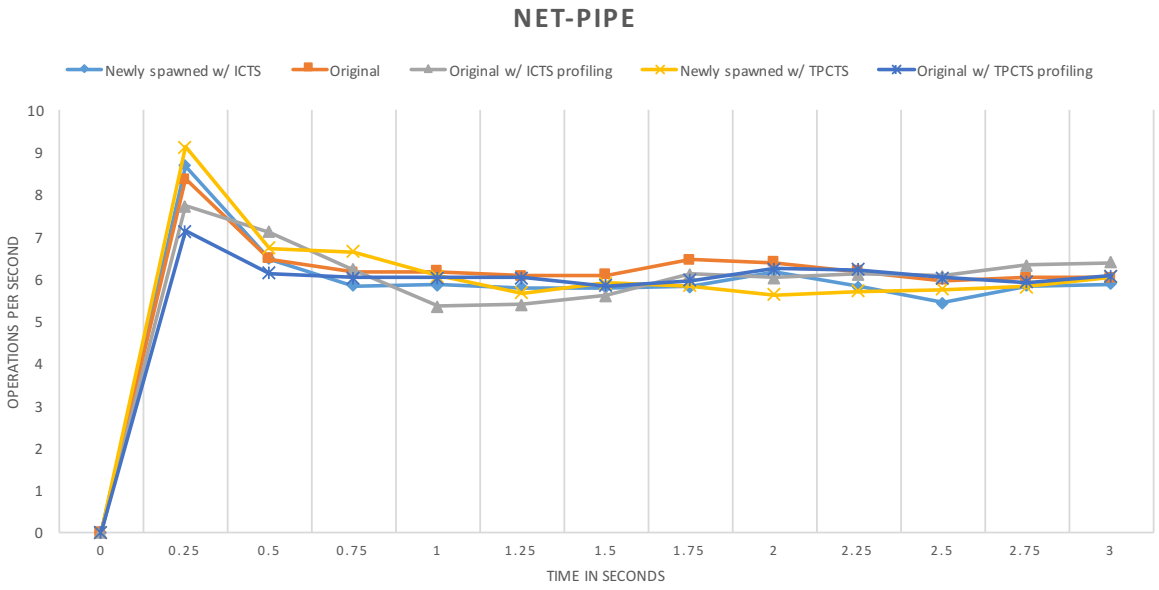

Figure 5.11: Operations per second plots for net-pipe benchmark. The y axis represents the number of API operations performed by the benchmark per second. The x axis represents execution time in seconds. Higher is better.

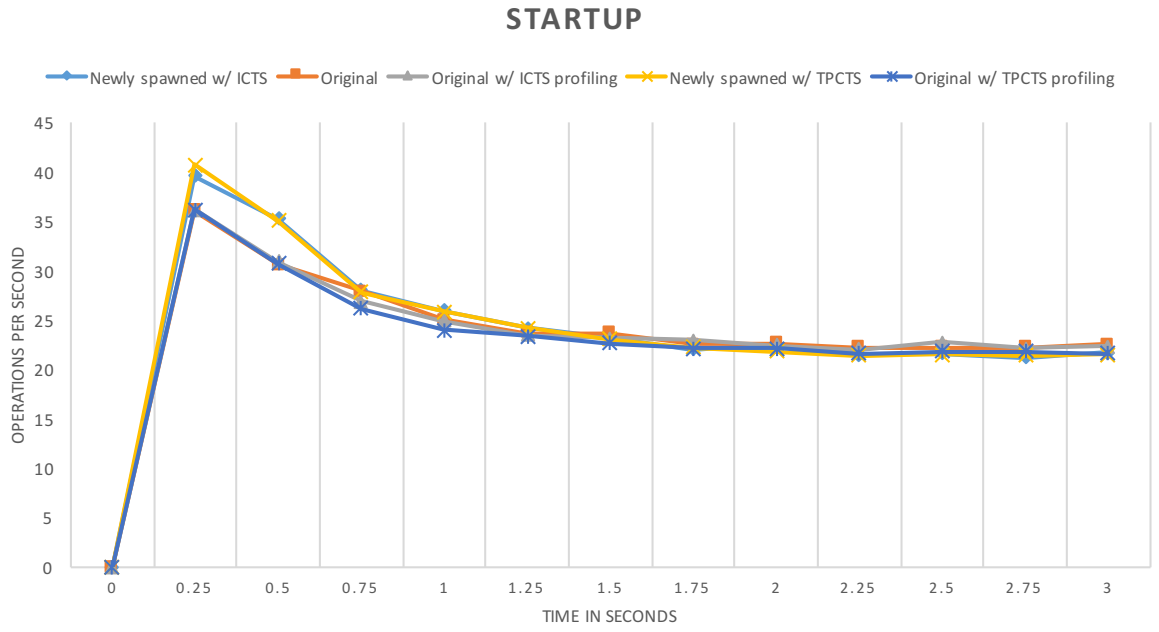

Figure 5.12: Operations per second plots for startup.js benchmark. The y axis represents the requests the number of API operations performed by the benchmark per second. The x axis represents execution time in seconds. Higher is better.

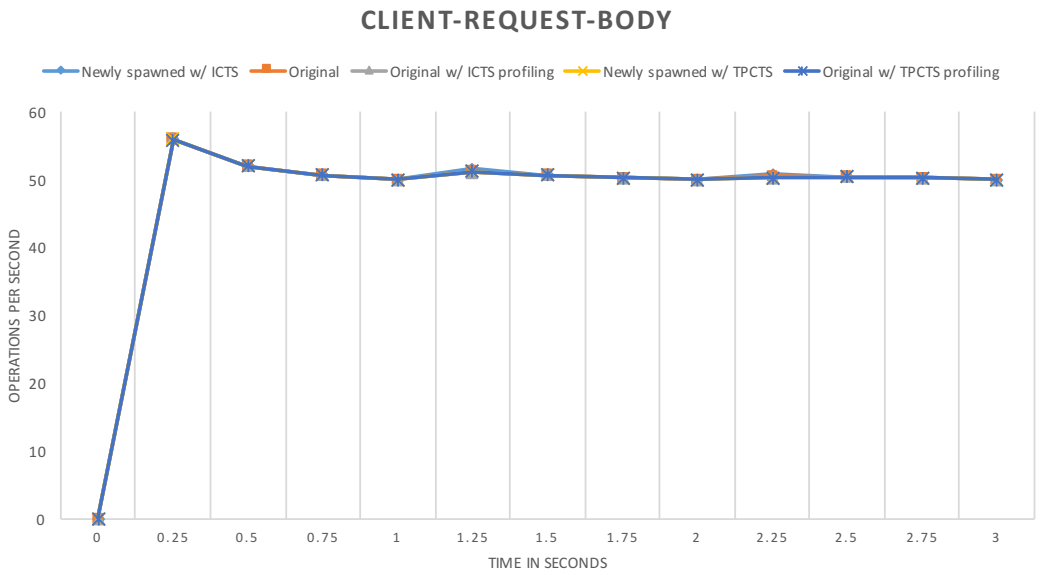

Figure 5.13: Operations per second plots for client-request-body.js benchmark. The y axis represents the requests the number of API operations performed by the benchmark per second. The x axis represents execution time in seconds. Higher is better.

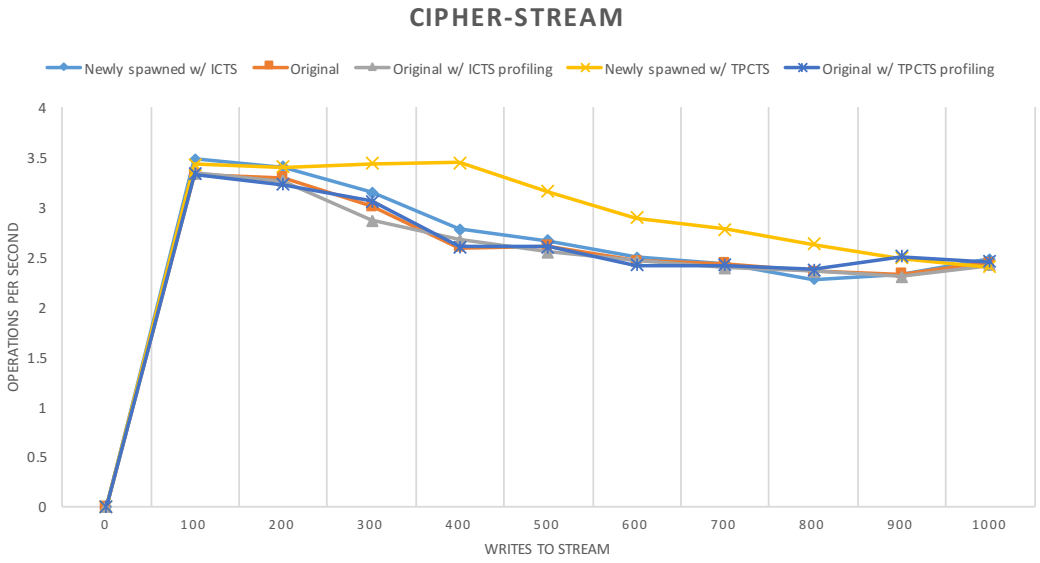

Figure 5.14: Operations per second plots for cipher-stream.js. The x axis represents the number of write operations performed by the server into the stream when requested by the client. The y axis represents the rate at which the write operations are performed per second. Higher is better.

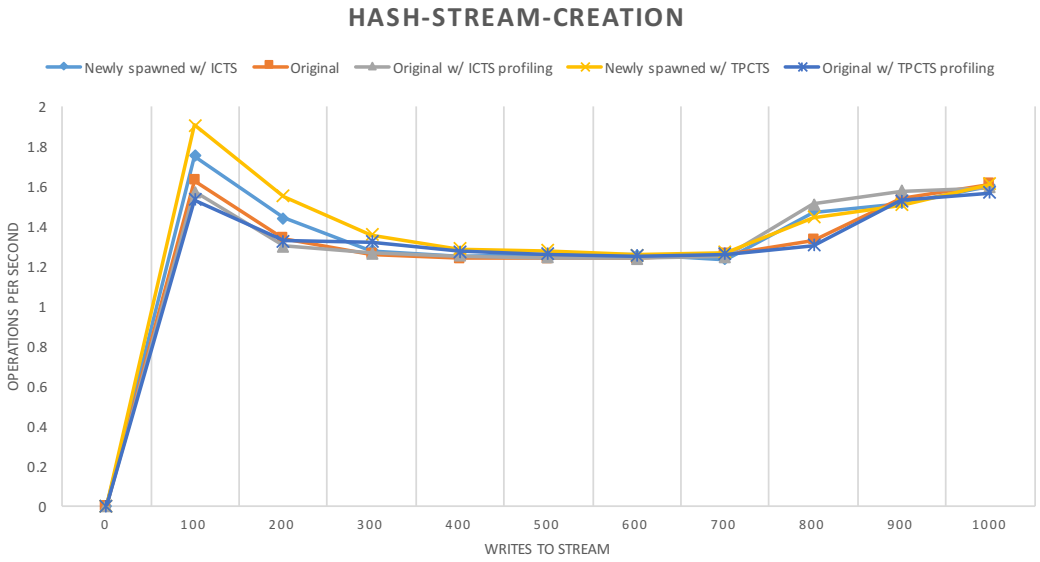

Figure 5.15: Operations per second plots for hash-stream-creation.js. The x axis represents the number of write operations performed by the server into the stream when requested by the client. The y axis represents the rate at which the write operations are performed per second. Higher is better.

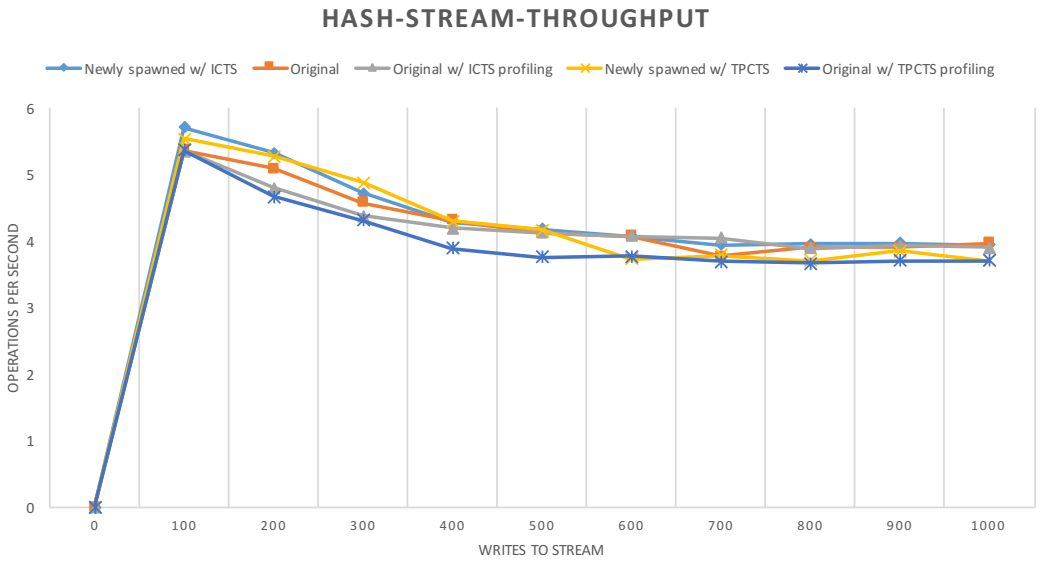

Figure 5.16: Operations per second plots for hash-stream-throughput.js. The x axis represents the number of write operations performed by the server into the stream when requested by the client. The y axis represents the rate at which the write operations are performed per second. Higher is better.

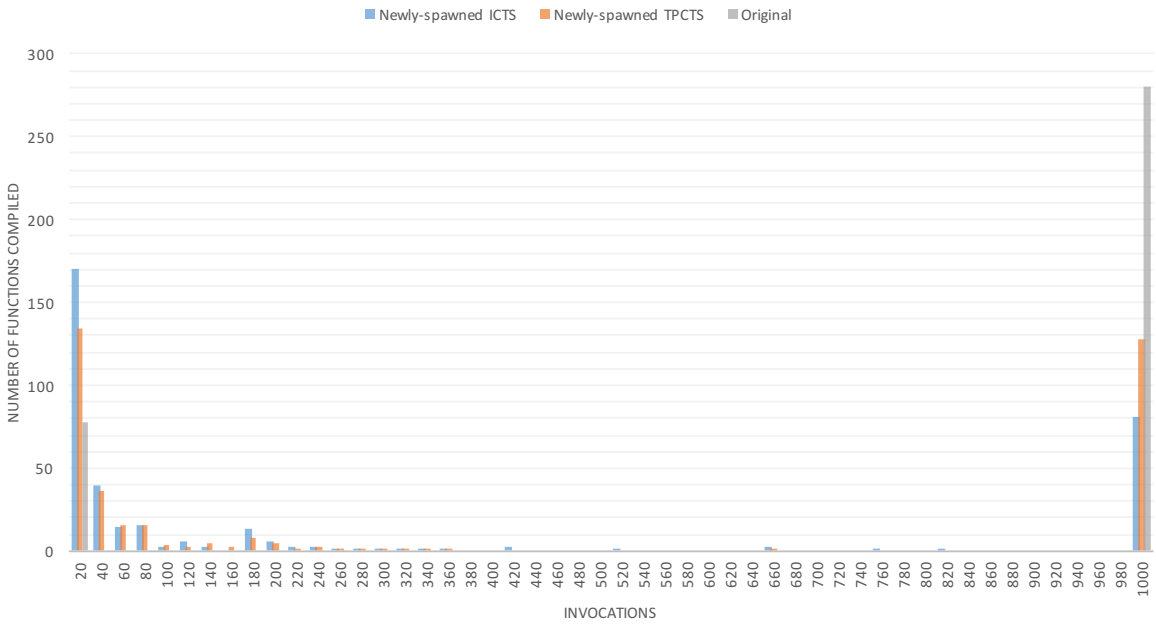

Figure 5.17: Number of functions compiled by the original engine, newly-spawned engine with ICTS and newly-spawned engine with TPCTS at every interval of 20 invocations.

spawned engine with ICTS profiling, most of the functions are compiled between 0 - 20 invocations. This is because most of the functions are type stable within the first twenty iterations. Some exceptions to this include a few library functions that are designed to deal with objects of different shapes and object types. These functions are compiled only after one thousand iterations.

In the newly-spawned engine with TPCTS profiling, most of the functions are compiled between 0–20 invocations. However, the number of functions compiled in this duration is fewer compared to the newly-spawned engine with ICTS. This is primarily because of the phase change problem described above. Therefore, a major chunk of functions are compiled only after one thousand iterations.

5.8 Conclusion

In this chapter we have addressed the problem of optimizing server-side JavaScript applications. Specifically, we modify the `node.js` runtime to transfer information collected from the original JavaScript engine to any newly spawned engines that the `node.js` runtime has created due to increased load. This information is used by the newly spawned engines to hasten the time at which functions can be compiled by the optimizing compiler; the result is an increased initial throughput.

We have identified the profile information collected by the original engine that can be easily transferred to the spawned engines. Faced with important profile information that *cannot* be easily transferred, we have identified a notion of function type stability that allows the original engine to transfer useful information to the spawned engines even without being able to transfer the exact profile information from the original engine. Our results show that for 7 out of 10 benchmarks, our technique shows better performance than the original engine. The overhead of collection of additional profile information in the original engine is negligible. Also, using the type stability metrics, the runtime engine can detect and optimize type stable functions earlier than the original engine.

Chapter 6

Conclusion

We conclude the dissertation by summarizing our key contributions to the field of type specialization and describing the limitations and future directions of research in this field.

Type specialization is an important optimization used to improve the performance of JavaScript engines. In this dissertation we present various techniques to augment and improve upon the current state-of-the-art type specialization techniques. Our work is a significant contribution to the JavaScript engine design community and opens up new avenues to push the limits of performance for JavaScript engines.

JavaScript engine design and optimization as a field is rapidly advancing with major corporations like Google, Apple, Microsoft, Oracle, and Mozilla having major stakes it. These corporations have their versions of JavaScript engines either embedded in web browsers or used as a language runtime in a server. In any case, these products affect the lives of millions of people everyday. We believe that the strategies described in this dissertation will further help improve the state-of-the-art in this field. As an example, this dissertation presents the *first* deoptimization technique that is implemented on top of a typed stack-based virtual machine. This technique is now implemented in Oracle's Nashorn JavaScript engine, which is the default JavaScript implementation bundled with

Java 8 distribution.

6.1 Contributions and Future Directions

We list the contributions of the work described in this dissertation and discuss the possible future directions for each of them.

6.1.1 Synergistic Type Specialization

We first solve the problem of combining type feedback and type inference to assist and augment each other. We improve upon the previous work of using type feedback to improve the precision of type inference by extending using function type signature. We use type inference analysis to reduce the overhead of type feedback by using inferred type information to intelligently place type profiling hooks This also helps in reducing the number of type checks that are performed during the optimized code execution. Via experimental evaluation we show that synergistic type specialization outperforms the state-of-the-art type specialization techniques across a wide array of benchmark suites.

Future Directions: In synergistic type specialization, we perform intra-procedural type inference. An obvious way to improve the precision of the analysis is to incorporate object types and function return types as additional inputs to the analysis. Since the analysis is performed online, there is an urgency to complete the analysis as early as possible. Therefore, the challenge is to balance the precision of the analysis with the time taken to perform the analysis.

The algorithm at the moment does not perform redundant guard elimination analysis. For example, the type checks or guards that are added around every access of a global variable are unnecessary if the variable is not modified between checks. In theory,

the analysis seems straightforward; but in practice, it is not non-trivial. The implicit conversion rules and the possibility of execution of arbitrary code while accessing global variables and object properties using `getters` and `setters` makes the analysis a lot more complicated. Therefore, if the JavaScript engine is designed to keep track of the access pattern of global variables and object properties, it is possible to perform redundant guard elimination.

JavaScript has evolved rapidly over the past couple of years. New features like typed arrays [94], proxies [95], classes [96], constants [97], and block scoping mechanism [98] have been introduced in ECMAScript 6 standard [99]. Making the synergistic type specialization technique aware of such features will be a challenge and will potentially improve the precision of the types that are inferred by it.

6.1.2 Deoptimization on Top of Typed, Stack-based Virtual Machines

Previous work on type specialization on top of typed, stack-based VMs implemented in runtimes like IronJS [10] do not use deoptimization as a recovery mechanism. This is due to inherent limitation of typed, stack-based runtimes that disallow pre-existing deoptimization techniques to work on top of them. We solve this problem by designing a new deoptimization strategy tuned to operate on top of typed, stack-based VMs. The strategy uses the exception handling feature of the VM and a new bytecode verifier, implemented on top of the VM. Our experimental evaluation shows that our technique performs better than pre-existing type specialization techniques implemented on top of typed, stack-based VMs.

Future Directions: One weakness of the deoptimization technique is that the algorithm assumes that, at deoptimization point, the values present in the operand stack can be converted into DValues that are processed by the subroutine threaded interpreter. In

some scenarios, the values present in the operand stack of the VM cannot be converted to DValues. For example, some optimizations such as polymorphic inline caching stores the `map` or `hidden class` of the object in the operand stack. Deoptimization is not possible at this point because the runtime is not able to convert the `map` structure into DValue.

Though we avoid type specializing functions containing such program points, this is still a limitation of the algorithm and can potentially limit new types of optimizations that can be applied on the code in the future. One solution to this problem is to maintain an auxiliary type stack during the code generation phase that tracks only those values that need to be transferred to the subroutine-threaded code. For any non-transferable values such as object maps, it is important to maintain a placeholder to skip it.

Our technique makes heavy use of exception handling to perform state transfer from optimized code to non-optimized code. Microsoft's MSDN library documentation suggests that throwing an exception explicitly in a try block can inhibit the compiler from performing certain optimizations [100]. One approach to solving this problem is by eliminating the need for exception handling while transferring the control from optimized code to non-optimized code. One possible alternative is to add support for deoptimization in the underlying VM itself by adding a special bytecode that indicates that deoptimization is possible at certain program points. The language implementor can provide a mapping from a deoptimization point in the optimized code to a corresponding program point in the non-optimized code where the execution should resume. The runtime can take care of the transferring the operand stack from optimized code to the non-optimized code.

6.1.3 Server-Side Type Profiling

Deoptimizations are an important concern for performance, and reducing the deoptimizations provides a significant performance benefit. We present a technique to optimize JavaScript programs sent from a server to a client by performing ahead-of-time profiling on the server side to reduce deoptimizations. In addition to reducing deoptimizations, our technique also allows aggressive compilation of hot functions, while reducing the change of deoptimizations caused due to reduced profiling time.

Future Directions: Researchers have made various attempts at adding static typing to JavaScript to improve programmer productivity and in some cases, to avoid type errors during run-time. Microsoft's TypeScript [67] language and Facebook's Flow type checker [65] are two popular technologies at the moment.

In case of TypeScript, the type annotations are discarded by the compiler when it translates TypeScript code to JavaScript. Therefore, the annotations only serve the purpose of ensuring the correctness of the code when it is compiled. Flow type checker performs type inference on JavaScript and uses type annotations, when present, to check for the correctness of the code. But when the code is compiled down to JavaScript all the type annotations are either lost or made explicit as type checks in the code itself using the `instanceOf` operator. Type checks performs as part of the code are slower and serve only the purpose of checking the code for correctness.

A possible extension to our work is to combine the type annotations added by the developer with the types that are collected during the ahead-of-time profiling phase to generate much precise type profile information. The developer can then be provided with an interface that shows where deoptimization occurred during the course of execution and what unexpected type caused it. With the help of the developers' input, the code can be modified to either eliminate the deoptimization or add additional annotations to the

code to improve the performance.

One of the drawbacks of server-side profiling is that not all classes of deoptimizations can be captured by ahead-of-time profiling. For example, a few deoptimizations occur due to unhandled corner cases in optimizations performed by the optimizing compiler. These optimizations and corresponding deoptimizations can be different for different client JavaScript engines. Sometimes the root cause of deoptimization is not made explicit by the client JavaScript engine. One way to solve this problem is to modify the client browsers to make the cause of deoptimization explicit. This will allow us to modify the ahead-of-time profiler to be much smarter in capturing the required information during deoptimizations.

Object shape tracking is still a major hurdle while performing ahead-of-time profiling. Our attempt at object shape tracking involved using the calling context of the program point where the shape was created to uniquely identify the shape. This turned out to be extremely expensive process both while profiling at the server-side and using the profile information at the client-side. This is mainly because the calling contexts used to identify object shapes were imprecise and tracking precise calling contexts while executing the code is expensive. A possible solution to this problem is to modify the JavaScript engine to explicitly track precise calling contexts in an efficient manner.

6.1.4 Accelerating Server-Side JavaScript

We identify the profile information that can be transfered from one server-side JavaScript engine to another to improve performance. We classify such profile information into two categories – cachable and heap-dependent information. Though transferring cachable information is trivial, heap-dependent information is not easily transferable. Therefore, we propose a notion of function type stability that indicates the newly spawned engines

when a function has collected enough heap-dependent type information to be optimized.

We evaluate our technique on Jxcore, a node.js implementation and compare the performance against ten node.js benchmarks. Our results show that seven out of ten benchmarks benefit from additional profile information and type stability heuristics. Additionally, the overhead of collection of additional profile information in the original engine is negligible.

Future Directions: The type stability heuristics are not always perfect. More precise the type stability heuristic is, better the chance of avoiding deoptimization in the newly spawned engine. An alternate and more accurate heuristic of type stability is based on tracking object shapes. Object shapes are the most predominant heap-dependent profile information. Therefore, as a type stability metric we can approximately consider a function to be type stable if for all program points in the function, all the previously recorded object shapes were observed in the newly spawned instances of the application. As described in the previous section, accurately tracking object properties is non-trivial and expensive. If the JavaScript engines can be modified to implement accurate and inexpensive calling context recording, the problem on object shape tracking can be solved. This will enable the a more accurate type stability metric based on object shape tracking.

Sometimes, inaccurate type stability heuristics can exasperate already existing problems in the applications. For example, in two of the benchmarks that we experimented with, type stability heuristics caused the garbage collector to discard the optimized code more frequently causing slowdowns. Therefore, it is important to track such adverse effects of using type stability heuristics and take remedial measures like resetting them to default runtime behavior.

Applications are known to evolve over time and phase changes while executing an application is well documented in literature [101, 50]. With phase change, profile infor-

mation often tends to become obsolete. One way to solve this problem for cached profile information is to to implement the concept of *profile decay* to phase out irrelevant information form the profile database. Another approach to handle phase changes is to add a temporal dimension to the profile information that is collected. For example, if a function is observed to operate on different set of types after a phase change, the new types can be recorded in the profile database along with the phase change indicator. Examples of phase change indicators for JavaScript applications are deoptimizations and creation of new object shapes during the course of execution of the application. Using this temporal dimension in the profile database, on phase change, the newly spawned JavaScript engine can trigger the optimizing compiler to generate fresh versions of optimized code.

Bibliography

[1] "Node.js JavaScript Runtime." https://nodejs.org/en/, 2015.

[2] "Windows 10 universal app platform." https://blogs.windows.com/buildingapps/2015/03/02/a-first-look-at-the-windows-10-universal-app-platform/, 2015.

[3] "Gnome javascript applications." https://developer.gnome.org/gnome-devel-demos/stable/js.html.en, 2015.

[4] "Javascript game engine." http://www.cocos2d-x.org/, 2015.

[5] "Espruino - javascript for microcontrollers." http://www.espruino.com/, 2015.

[6] "Tessel 2 - javascript-based development platform for microcontrollers." https://tessel.io/, 2015.

[7] C. Chambers, J. Hennessy, and M. Linton, *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages*, tech. rep., Stanford University, Department of Computer Science, 1992.

[8] B. Hackett and S.-y. Guo, *Fast and precise hybrid type inference for javascript*, in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 239–250, ACM, 2012.

[9] M. Bebenita *et. al.*, *SPUR: a trace-based JIT compiler for CIL*, in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, (Reno/Tahoe, Nevada, USA), pp. 708–725, 2010.

[10] "IronJS." https://github.com/fholm/IronJS.

[11] "Rhino JavaScript engine." https://developer.mozilla.org/en-US/docs/Rhino.

[12] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu, *On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages*, in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 195–212, ACM, 2012.

[13] "Nashorn JavaScript engine." `http://openjdk.java.net/projects/nashorn`.

[14] "SpiderMonkey JavaScript Engine." `http://www.mozilla.org/js/spidermonkey/`, 2015.

[15] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf, *Improved type specialization for dynamic scripting languages*, in *Proceedings of the 9th Symposium on Dynamic Languages*, 2013.

[16] "Acm author rights." `http://authors.acm.org/main.html`, 2015.

[17] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf, *Deoptimization for dynamic language jits on typed, stack-based virtual machines*, in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.

[18] M. N. Kedlaya, B. Robatmili, and B. Hardekopf, *Server-side type profiling for optimizing client-side javascript engines*, in *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, (New York, NY, USA), pp. 140–153, ACM, 2015.

[19] U. Hölzle and D. Ungar, *Optimizing dynamically-dispatched calls with run-time type feedback*, *ACM SIGPLAN Notices* **29** (1994), no. 6 326–336.

[20] O. Agesen, *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA, 1996. UMI Order No. GAX96-20452.

[21] O. Agesen and U. Hölzle, *Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages*, in *ACM SIGPLAN Notices*, vol. 30, pp. 91–107, ACM, 1995.

[22] "Sunspider benchmark suite." `http://www.webkit.org/perf/sunspider/sunspider.html`.

[23] "V8 benchmark suite." `http://v8.googlecode.com/svn/data/benchmarks/v7/README.txt`.

[24] "Kraken benchmark suite." `http://krakenbenchmark.mozilla.org`.

[25] "Js1k." `http://js1k.com`.

[26] U. Hölzle and D. Ungar, *Reconciling responsiveness with performance in pure object-oriented languages*, *ACM Trans. Program. Lang. Syst.* **18** (July, 1996) 355–400.

[27] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, *A survey of adaptive optimization in virtual machines*, in *Proceedings of the IEEE, 93(2), 2005. special issue on program generatation, optimization, and adaptations*, 2004.

[28] M. Arnold and B. G. Ryder, *A framework for reducing the cost of instrumented code*, in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, (New York, NY, USA), pp. 168–179, ACM, 2001.

[29] "Google Inc. V8 JavaScript virtual machine." `https://code.google.com/p/v8`, 2015.

[30] "V8 engine." `http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler`, 2013.

[31] C. Chambers, *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages.* PhD thesis, Stanford University, 1992.

[32] U. Hölzle, *Adaptive optimization for SELF: reconciling high performance with exploratory programming.* PhD thesis, Stanford University, 1995.

[33] 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*, p. 9, ACM, 2011.

[34] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, *Tracing the meta-level: Pypy's tracing jit compiler*, in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25, ACM, 2009.

[35] "PyPy Status Blog." `http://morepypy.blogspot.com`, 2013.

[36] "Rubinius Blog." `http://rubini.us/blog`, 2013.

[37] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, *Common lisp object system specification*, *ACM Sigplan Notices* **23** (1988), no. SI 1–142.

[38] C. Chambers and G. T. Leavens, *Typechecking and modules for multimethods*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **17** (1995), no. 6 805–843.

[39] "Xamarian Inc. Mono." http://www.mono-project.com/Main_Page, 2013.

[40] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar, *Zoomm: a parallel web browser engine for multicore mobile devices*, in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, (New York, NY, USA), pp. 271–280, ACM, 2013.

[41] "Google closure compiler." https://developers.google.com/closure/compiler.

[42] "Jscrush minifier." http://www.iteral.com/jscrush.

[43] S. J. Fink and F. Qian, *Design, implementation and evaluation of adaptive recompilation with on-stack replacement*, in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003.

[44] S. Soman and C. Krintz, *Efficient and general on-stack replacement for aggressive program specialization*, in *Proceedings of the 2006 International Conference on Programming Languages and Compilers*, 2006.

[45] U. Hölzle and D. Ungar, *A third-generation self implementation: reconciling responsiveness with performance*, in *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, 1994.

[46] M. Paleczny, C. Vick, and C. Click, *The java hotspot server compiler*, in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, 2001.

[47] B. Robatmili, C. Cascaval, M. Reshadi, M. N. Kedlaya, S. Fowler, M. Weber, and B. Hardekopf, *Muscalietjs: Rethinking layered dynamic web runtimes*, in *Proceedings of the 10th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2014.

[48] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown, *Context threading: A flexible and efficient dispatch technique for virtual machine interpreters*, in *Proceedings of the international symposium on Code generation and optimization*, 2005.

[49] "Dynamic Language Runtime." http://msdn.microsoft.com/en-us/library/dd233052.aspx.

[50] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, *Trace-based just-in-time*

type specialization for dynamic languages, in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[51] "Lua Just-In-Time compiler." `http://luajit.org/`, 2013.

[52] S. Brunthaler, *Efficient interpretation using quickening*, in *Proceedings of the 6th Symposium on Dynamic Languages*, 2010.

[53] S. Brunthaler, *Inline caching meets quickening*, in *Proceedings of the 24th European Conference on Object-oriented Programming*, 2010.

[54] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani, *Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs*, in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[55] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck, *An intermediate representation for speculative optimizations in a dynamic compiler*, in *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, 2013.

[56] U. Hölzle, C. Chambers, and D. Ungar, *Debugging optimized code with dynamic deoptimization*, in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, 1992.

[57] M. Arnold, A. Welc, and V. T. Rajan, *Improving virtual machine performance using a cross-run profile repository*, in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.

[58] C. Krintz, *Coupling on-line and off-line profile information to improve program performance*, in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2003.

[59] "Java virtual machine." `http://docs.oracle.com/javase/specs/jvms/se7/html/`, 2013.

[60] "MSDN. (2011, March) Common Language Runtime Overview." `http://msdn.microsoft.com/en-us/library/ddk909ch.aspx`.

[61] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, *Adaptive optimization in the jalapeno jvm*, in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[62] "Webkit JavaScriptCore virtual machine." `http://trac.webkit.org/wiki/JavaScriptCore`, 2015.

[63] C. Krintz and B. Calder, *Using annotations to reduce dynamic optimization time*, in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[64] "asm.js specification." `http://asmjs.org/spec/latest/`, 2015.

[65] "Flow static type checker." `http://flowtype.org`, 2015.

[66] "Google Inc. closure compiler." `https://developers.google.com/closure/compiler/`, 2015.

[67] "Typescript." `http://www.typescriptlang.org`, 2015.

[68] L. Guckert, M. OConnor, S. Kumar Ravindranath, Z. Zhao, and V. Janapa Reddi, *A case for persistent caching of compiled javascript code in mobile web browsers*, in *In Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2013.

[69] J. Oh and S.-M. Moon, *Snapshot-based loading-time acceleration for web applications*, in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[70] "Spidermonkey baseline compiler." `https://wiki.mozilla.org/Javascript:SpiderMonkey:BaselineCompiler`, 2013.

[71] U. Hölzle, C. Chambers, and D. Ungar, *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*, in *ECOOP'91 European Conference on Object-Oriented Programming*, pp. 21–38, Springer, 1991.

[72] "Octane benchmark suite." `https://developers.google.com/octane/`, 2015.

[73] "pixi.js 3d rendering engine." `http://www.pixijs.com`, 2015.

[74] "three.js physics engine." `http://threejs.org`, 2015.

[75] "Matter.js - a 2d rigid body javascript physics engine." `http://brm.io/matter-js/`, 2015.

[76] "Physics.js physics engine." `http://wellcaffeinated.net/PhysicsJS/`, 2015.

[77] "Membench50." `http://gregor-wagner.com/tmp/mem50`, 2015.

[78] "Mozilla central repository." `https://hg.mozilla.org/mozilla-central`, 2015.

[79] "Selenium ide." http://docs.seleniumhq.org/docs/02_selenium_ide.jsp, 2015.

[80] "Jxcore Node.js Distribution." http://jxcore.com/home/, 2015.

[81] "Avatar.js Node.js Distribution." https://avatar-js.java.net/, 2015.

[82] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, *Microarchitectural implications of event-driven server-side web applications*, in *Proceedings of International Symposium on Microarchitecture*, 2015.

[83] "Nodejs require is dog slow." https://kev.inburke.com/kevin/node-require-is-dog-slow/, 2015.

[84] "From Node.js To Go." http://thenewstack.io/from-node-js-to-go-why-one-startup-made-the-switch/, 2015.

[85] "A little problem with child process." https://goo.gl/dD8WRF, 2015.

[86] "When require is slow." https://goo.gl/LM1uEf, 2015.

[87] "nwjc Snapshot Tool." https://github.com/nwjs/nw.js/wiki/Protect-JavaScript-source-code-with-v8-snapshot, 2015.

[88] "Nginx reverse proxy server." https://www.nginx.com/, 2015.

[89] "HAProxy load balancer." http://www.haproxy.org/, 2015.

[90] "Varnish load balancer." https://www.varnish-cache.org/, 2015.

[91] "Cluster Node.js module." https://nodejs.org/docs/v0.6.0/api/cluster.html, 2015.

[92] M. W. Stephenson, *Automating the construction of a complier heuristics using machine learning*, *Thesis (Ph. D.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science* (2006).

[93] S. M. Sandya, *Jazzing up jvms with off-line profile data: Does it pay?*, *SIGPLAN Not.* **39** (Aug., 2004) 72–80.

[94] "ECMAScript typed arrays." http://www.khronos.org/registry/typedarray/specs/latest/.

[95] "Javascript proxies." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy, 2015.

[96] "Javascript classes." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, 2015.

[97] "Javascript constants." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const, 2015.

[98] "Javascript block scoping." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let, 2015.

[99] "Ecmascript 6 features." http://es6-features.org/, 2015.

[100] "Visual c++ optimization best practices." https://msdn.microsoft.com/en-us/library/ms235601.aspx, 2015.

[101] P. Nagpurkar and C. Krintz, *Visualization and analysis of phased behavior in java programs*, in *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, PPPJ '04, pp. 27–33, Trinity College Dublin, 2004.