Université de Montréal

On the Fly Type Specialization without Type Analysis

par Maxime Chevalier-Boisvert

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Décembre, 2015

RÉSUMÉ

Les langages de programmation typés dynamiquement tels que JavaScript et Python repoussent la vérification de typage jusqu'au moment de l'exécution. Afin d'optimiser la performance de ces langages, les implémentations de machines virtuelles pour langages dynamiques doivent tenter d'éliminer les tests de typage dynamiques redondants. Cela se fait habituellement en utilisant une analyse d'inférence de types. Cependant, les analyses de ce genre sont souvent coûteuses et impliquent des compromis entre le temps de compilation et la précision des résultats obtenus. Ceci a conduit à la conception d'architectures de VM de plus en plus complexes.

Nous proposons le versionnement paresseux de blocs de base, une technique de compilation à la volée simple qui élimine efficacement les tests de typage dynamiques redondants sur les chemins d'exécution critiques. Cette nouvelle approche génère paresseusement des versions spécialisées des blocs de base tout en propageant de l'information de typage contextualisée. Notre technique ne nécessite pas l'utilisation d'analyses de programme coûteuses, n'est pas contrainte par les limitations de précision des analyses d'inférence de types traditionnelles et évite la complexité des techniques d'optimisation spéculatives.

Trois extensions sont apportées au versionnement de blocs de base afin de lui donner des capacités d'optimisation interprocédurale. Une première extension lui donne la possibilité de joindre des informations de typage aux propriétés des objets et aux variables globales. Puis, la spécialisation de points d'entrée lui permet de passer de l'information de typage des fonctions appellantes aux fonctions appellées. Finalement, la spécialisation des continuations d'appels permet de transmettre le type des valeurs de retour des fonctions appellées aux appellants sans coût dynamique. Nous démontrons empiriquement que ces extensions permettent au versionnement de blocs de base d'éliminer plus de tests de typage dynamiques que toute analyse d'inférence de typage statique.

Mots clés: compilation à la volée, optimisation, typage dynamique, code, analyse, performance.

ABSTRACT

Dynamically typed programming languages such as JavaScript and Python defer type checking to run time. In order to maximize performance, dynamic language virtual machine implementations must attempt to eliminate redundant dynamic type checks. This is typically done using type inference analysis. However, type inference analyses are often costly and involve tradeoffs between compilation time and resulting precision. This has lead to the creation of increasingly complex multi-tiered VM architectures.

We introduce *lazy basic block versioning*, a simple just-in-time compilation technique which effectively removes redundant type checks from critical code paths. This novel approach lazily generates type-specialized versions of basic blocks on the fly while propagating context-dependent type information. This does not require the use of costly program analyses, is not restricted by the precision limitations of traditional type analyses and avoids the implementation complexity of speculative optimization techniques.

Three extensions are made to the basic block versioning technique in order to give it interprocedural optimization capabilities. *Typed object shapes* give it the ability to attach type information to object properties and global variables. *Entry point specialization* allows it to pass type information from callers to callees, and *call continuation specialization* makes it possible to pass return value type information back to callers without dynamic overhead. We empirically demonstrate that these extensions enable basic block versioning to exceed the capabilities of static whole-program type analyses.

Keywords: JIT, VM, compiler, dynamic typing, optimization, code, analysis, performance.

CONTENTS

LIST OF APPENDICES

LIST OF ABBREVIATIONS

AOT	Ahead-Of-Time (before a program is executed)
AST	Abstract Syntax Tree
BBV	Basic Block Versioning
CPU	Central Processing Unit
DOM	Document Object Model
ES5	ECMAScript version 5 (JavaScript specification)
GC	Garbage Collector
HTML	HyperText Markup Language
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
JIT	Just-In-Time (or just-in-time compiler)
JS	The JavaScript programming language
ML	Machine Learning
NaN	The special Not-a-Number floating-point value
OSR	On-Stack Replacement
PIC	Polymorphic Inline Cache
RAM	Random Access Memory
SSA	Static Single Assignment form
VM	Virtual Machine
V8	Google V8 JavaScript virtual machine

PREFACE

I discovered the world of dynamic language optimization during my master's degree at McGill University, under the supervision of Prof. Laurie Hendren. My thesis was to be focused on the design and implementation of McVM, an optimizing virtual machine for the MATLAB programming language [9]. I believe what Prof. Hendren had in mind, when she suggested I take on this project, was that I would work on compiler optimizations specifically related to numerical computing.

What really struck me, instead, is that MATLAB is a dynamically typed, late-bound programming language. The semantics of the language imply that variables can change type at run time, and that dynamic type tests are needed to implement the bulk of operations. It seemed intuitively obvious to me that there had to be a way to eliminate much of these type tests, since in practice, in a given program, most variables do not change type. It was necessary to achieve this in order to obtain good performance, but none of the static, AOT program analysis techniques I'd read about seemed adequate. It was an unsolved problem, and this greatly stimulated my curiosity.

JIT compilers generate code at run time. This can be seen as limiting, because the time budget for compiling and optimizing code is smaller than with an AOT compiler. The flip side of this, however, is that JIT compilers have an amazingly powerful tool at their disposal: they can observe running programs while optimizing them. The key insight of my master's thesis was that it's possible to leverage this ability, in a small way, to optimize dynamically typed programs. By lazily compiling methods and intercepting their argument types, one can generate type-specialized versions of the said methods. McVM was built around this principle, and we were able to outperform the MathWorks MATLAB compiler on several benchmarks.

In 2009, I went on to do a Ph.D. at the Université de Montréal. I didn't go into compiler research, but instead joined the machine learning lab. I loved compilers and programming languages, but I had convinced myself that I should be working in ML as it seemed like a more important field for the future of mankind, or something to that effect. Unfortunately, it became obvious to me after just a few weeks that this was the

wrong decision. I was having difficulty finding motivation to work on ML research, and I found myself spending a lot of time thinking about McVM. I had drawn up a long list of improvements I could make upon my master's research. This is when I decided to rearch out to Prof. Marc Feeley.

I came to see Prof. Feeley with the goal of working on a hybrid type analysis for dynamic languages. That is, a type analysis that's able to speculate and update itself as a program runs, interleaving execution and analysis. Prof. Feeley was somewhat skeptical of the concept, but was also eager to have me onboard and willing to give this project a chance. It was decided that we would implement a JavaScript VM, in large part because it was obvious that this programming language was rapidly growing in popularity, but also because its specification was relatively simple.

My PhD has taken over six years to complete, and it's not been a walk in the park. I've seriously considered giving up on more than one occasion, but I persevered because I love what I do. I like research, and I find this work interesting. As Ernest Hemingway said, "It is good to have an end to journey toward; but it is the journey that matters, in the end." Along the way, I've learned countless things, published my research, been invited to speak about my work at five industry conferences, started making a name for myself and received a job offer before I'd even graduated. In the end, I believe that the effort was worth it. I'm proud of what I've accomplished.

Maxime Chevalier-Boisvert, Montreal, February 1st, 2016

ACKNOWLEDGMENTS

Thanks to my thesis advisor, Marc Feeley, for putting up with my stubbornness and for standing by me during difficult times. It took me over six years to complete my PhD, and it has been trying at times, but thanks to you, I am a better scientist.

Thank you Erinn Di Staulo, Catherine Gagnon, Christian Yelle and Jonathan Joseph for your help and emotional support throughout the last few years.

Thank you Laurie Hendren, Jan Vitek, Erick Lavoie, Vincent Foley, Paul Khuong, Carl Friedrich Bolz and Jacques Amar for the invaluable advice and help you've provided in preparing publications and completing my thesis.

Thanks to Molly Everett, Brett Fraley, Simon Bernier St-Pierre, Rajaram Gaunker, Gianluca Stivan, Olivier Matz and all of those who have contributed to the development of the Higgs VM.

Thanks to the D programming language community for being so welcoming and supportive of my work, as well as for the public exposure you have given my research.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Mozilla Corporation. Many Ph.D. candidates struggle to make ends meet during their studies, which can be a major cause of stress. Thanks to these organizations, I was never put in such a position, and I am very grateful for this.

CHAPTER 1

INTRODUCTION

The last two decades have seen a rise in the popularity of dynamically-typed programming languages such as Python, PHP and JavaScript, driven by the belief that features such as dynamic type checking and late-binding reduce the learning curve and vastly increase developer productivity. The establishment of JavaScript as the only language able to run in a web browser made it necessary to find ways to optimize dynamic languages. This is a challenging problem because features such as late-binding, dynamic type-checking, dynamic code loading and the infamous `eval` construct make AOT whole-program analyses largely inapplicable.

Recently, there have been advances in the use of hybrid type analyses in commercial JavaScript VMs. Unfortunately, these come with a few disadvantages. The first is that such analyses can be complex to implement, relying on mechanisms such as On-Stack Replacement (OSR). The second is that due to high compilation-time costs, these remain limited to long-running programs only. Lastly, as we will expand upon in later chapters, there are precision limitations inherent to traditional type analyses.

This thesis is an exploration of Basic Block Versioning (BBV), a new kind of JIT compiler architecture lying somewhere between method-based and trace compilation. With BBV, individual basic blocks are compiled one at a time and specialized based on accumulated type information. The technique relies on controlled tail duplication, that is, the selective versioning (cloning) of basic blocks, to extract and propagate type information.

One of the main strengths of the BBV technique is its conceptual simplicity and its relatively low overhead. It does not use profiling or type analysis in the traditional sense. Code is specialized on the fly, in a single pass, as it is generated, without the use of an iterative fixed point analysis. It is the execution of programs, and the code paths taken at run time, which drive the versioning and code specialization process.

The main metric we have focused on, throughout this work, is the number of dynamic

type tests performed with and without specific optimization techniques. Code is specialized in other ways besides the elimination of type tests, but this single metric gives us a good insight into how well our approach is doing in the realm of type-specialization.

Other metrics we have focused on throughout this research are machine code size and execution time. Code size is an indirect reflection of code quality. That is, code that is smaller in size usually contains fewer instructions, and so has been more effectively optimized. Our insistence on code size also stems from the need to deflect an obvious criticism of basic block versioning: the possibility of a code size explosion.

Execution time is probably the most often cited performance metric in compiler research, and successfully reducing it shows that our optimizations are working effectively. This is important because it is possible to reduce a metric such as the number of type tests executed without producing measurable performance gains. Reductions in execution time are the ultimate concrete proof that a performance optimization is effective.

1.1 Contributions

This thesis makes four main contributions.

1. The Basic Block Versioning (BBV) approach to compilation, and its use for type-specializing programs (Chapter 4).
2. The augmentation of BBV with the lazy, incremental compilation of basic blocks (Chapter 5).
3. The integration of typed shapes in a BBV system, by using BBV to generate versions based on object shapes (Chapter 6).
4. The addition of interprocedural type propagation strategies to BBV, making it possible to pass type information from callers to callees and back (Chapter 7).

1.2 Structure of this Document

Chapter 2 begins by introducing background material that is useful in understanding this thesis, but is not covered in depth as part of the articles included in subsequent

chapters. This includes a discussion of dynamic programming languages, JavaScript, and modern performance optimizations techniques for dynamic languages

Five articles submitted as part of this thesis are presented in Chapters 3 to 7. The first introduces the Tachyon self-hosted VM for JavaScript. The next four explain the BBV technique, as well as several improvements and additions made to it. Each article is preceded by a high-level discussion of its goals, contents and findings.

Additional experiments and additional data gathered about BBV but not included in published articles are presented in Chapter 8.

Finally, Chapter 9 provides an extensive survey into possible directions of future research and Chapter 10 presents some concluding remarks.

CHAPTER 2

BACKGROUND

This chapter discusses background information useful in understanding the research work presented in the following chapters. We will discuss dynamic programming languages, their origin and key features, the JS language and its role in our research, dynamic language optimization and state of the art JS VMs.

2.1 Dynamic Programming Languages

Dynamic programming languages are a loosely-defined family of programming languages incorporating dynamic typing, late binding, garbage collection and dynamic code loading (i.e. `eval`). This includes JavaScript, Python, Ruby, MATLAB, PHP, Scheme, Smalltalk and Perl. Dynamic languages can be compiled ahead of time, but modern implementations typically rely on an interpreter or JIT compiler instead.

The first dynamic language was LISP [31], originally specified by John McCarthy in 1958. This programming language originally executed in an interpreter and featured a *read-eval-print loop* (REPL), allowing programmers to enter code at a console and obtain immediate feedback. This was quite remarkable at the time, since much programming work was still being done using punched cards, which were cumbersome and error-prone.

LISP was inspired by the untyped lambda calculus [11], and introduced the concept of dynamic typing. Dynamic typing was likely motivated by the need to do away with type annotations to keep the notation short, making programming in a REPL more practical. Another motivation may have been that the static type systems of the time were cumbersome and weak. That is, when LISP was created, Hindley-Milner type inference [32] had not yet been invented. Static typing usually meant decorating all function arguments and variable declarations with type annotations and relying on unsafe type casting operators to circumvent limitations of the type system.

Garbage collection was also invented by John McCarthy for LISP. The language relied on linked list manipulation for computation. This meant that many temporary objects were allocated. Relying on the programmer to manually deallocate all these objects proved to be error-prone and tedious, particularly when programming in a REPL. Hence, LISP instead featured a copying Garbage Collector (GC) which would collect unreachable objects automatically.

With late binding, method names (both global and local) are resolved at the time of calls rather than ahead of time. This means methods can generally be redefined at any point during execution. This is in contrast with programming languages such as C, where the compiler resolves function calls at compilation time, and functions cannot be redefined during execution. Late binding poses some difficulty for optimizing compilers, as it means that the target of a given function call could change during the execution of a program.

Dynamic code loading refers to the ability to load and execute new source code at any point during a program's execution. This is typically embodied by the `eval` function, which allows evaluating the value of arbitrary strings of source code. The `eval` function was originally introduced by the LISP programming language, and gave programmers direct access to the LISP interpreter from within LISP programs. This function embodies one of the main tenets of LISP, which is that programming code is representable as data, and data can also be interpreted as code.

The presence of the `eval` construct makes dynamic languages notoriously difficult to optimize through program analysis, because its presence implies that, at any given time during a program's execution, parts of the program may be essentially unknown. That is, the `eval` function makes it possible to introduce arbitrary code which breaks properties previously proved by program analysis. As such, many ahead of time program analyses are unsound in the presence of the `eval` construct.

2.2 JavaScript

The research presented in this thesis should be applicable to any dynamic language. We have chosen to work with the JavaScript (JS) programming language (ECMAScript 5 [24]) because JS has the features typical of most dynamic languages, is widely-used in the real world, and has a fairly simple specification, which made it feasible for us to implement a JS compiler from scratch.

JavaScript is a dynamic programming language with object-oriented features and functional capabilities (i.e. nested functions and closures). It was originally created by Brendan Eich in 1995, while he was working on the Netscape Navigator web browser. The language was then meant to be a lightweight client-side scripting language that would compete with Microsoft's Visual Basic. Despite what the name suggests, JS is not directly related to the Java programming language.

The JS language features a few basic types: booleans, numbers, immutable strings, arrays, objects and closures. There are also two special constants: `null` and `undefined` which have special meaning. The language specification states that all JS numbers are double-precision floating-point values [24]. There is no distinct integer type in the language. This choice was made to ease the learning curve for beginners.

JS is dynamically typed. It features no type annotations and no compile-time type checking. Figure 2.1 shows a JS function in which both parameters can have various types in different contexts, and the return value can be either an integer or a boolean.

In dynamic languages such as JS, type errors only manifest themselves at execution time, when evaluating an operator with operands of inappropriate types. Some operations will produce an exception, but others may silently fail, that is, produce an erroneous value without an exception being thrown. For instance, reading an uninitialized variable or a non-existent object property will produce the special `undefined` value. Also, arithmetic operators in JS can accept all input types, but some combinations of input values are invalid and yield no numerical result (e.g. `undefined + 1`). These produce the special `NaN` (Not-a-Number) value.

Very few operations in JS throw exceptions on invalid inputs, most silently fail in-

```
/**
Function which returns the index of a value in an array-like
object. The function can accept arrays, strings, or anything
else with a 'length' and integer-keyed properties.
*/
function indexOf(array, value)
{
    for (var i = 0; i < array.length; ++i)
        if (array[i] == value)
            return i;

    // If value is not found, the boolean false is returned
    return false;
}

// Arrays can contain values of multiple types
var arr = ['a', 'b', 3, null];

indexOf(arr, 'b'); // returns 1
indexOf(arr, 4); // returns false

// Strings have a length property and can be indexed
// with integer indices to access their unicode code units
var str = 'foobar';

indexOf(str, 'b'); // returns 3
```

Figure 2.1: Arrays and dynamic typing in JS

stead. This complicates type analysis because we cannot make the assumption that invalid inputs will stop the normal flow of execution. For example, if a type analysis cannot prove that a given variable will never have the `undefined` value, then it must assume that numerical operators taking this variable as input may produce a `NaN` output. The `NaN` value can further propagate into other arithmetic operations, which could then produce `NaN` outputs in turn, polluting analysis results in a sort of chain reaction.

Some JS operations do throw exceptions. Notably, attempting to call a value which is not a function as if it were one will throw a `TypeError` exception, and so will attempting to read a property from a `null` or `undefined` base. In the example from Figure 2.1, attempting to evaluate `indexOf(null, 3)` will throw an exception when attempting to read `array.length`. However, passing an object without a `length` property to the `indexOf` function will cause it to silently fail and return `false`.

In JS, objects behave, to the programmer, as associative arrays mapping string keys (property names) to values. Properties of any type can be added to or removed from objects at any time. The language also features reflection capabilities, such as listing and iterating through the properties of objects. Figure 2.2 shows a small example where an object is used as a dictionary to map food and drink item names to price values.

```
var prices = {
    pizza: 10,
    coffee: 3,
    beer: 5,
    coke: 2
};

function getPrice(itemName)
{
    if (itemName in prices)
        return prices[itemName];

    throw Error('unknown␣item:␣' + itemName);
}

function addItem(itemName, price)
{
    if (itemName in prices)
        throw Error('duplicate␣item:␣' + itemName);

    if (typeof price != 'number' || !(price > 0))
        throw Error('price␣must␣be␣a␣positive␣number');

    prices[itemName] = price;
}
```

Figure 2.2: Objects as dictionaries in JS

JS objects follow a prototype-based inheritance model inspired from the Self [4, 35, 36] programming language. In this model, there are no classes. Instead, objects may derive from other objects in an acyclic prototype hierarchy. In effect, each object has a (potentially `null`) pointer to a prototype object. When a property is looked up on a given object and the property is not defined, then the same property lookup is recursively attempted on the prototype object until either an object defining the requested property is found along the prototype chain, or a `null` prototype pointer is encountered, in which

case the `undefined` value is produced.

Global variables in JS are stored on a first-class global object, which behaves like all other JS objects. This means that global function calls involve an implicit object property lookup, resulting in late binding. All JS functions in JS are implicitly variadic, that is one can pass any number of arguments to any function without an exception being thrown.

2.3 Early History of Dynamic Language Optimization

LISP, the first dynamic language, originally ran in an interpreter. The choice was made to value safety (reducing the risk of programmer error) and programmer productivity above performance. This was a sensible choice when it came to writing experimental programs, particularly for research purposes, but it hindered the adoption of languages such as LISP, particularly given that in the early 1970s, computational resources were scarce, limited and expensive.

It was obvious to John McCarthy that dynamic programming languages such as LISP could be made faster by compiling them to machine code instead of having them run in an interpreter. Because compilation took time, McCarthy had the idea that source-level annotations could be used to tell the LISP interpreter that specific functions, deemed by the programmer to be crucial for performance, were to be compiled instead of interpreted.

Even naive compilation of programs written in dynamic languages can easily provide large speedups over interpretation. However, this is not sufficient to get close to the performance levels achievable with statically-typed programming languages. Naive compilation eliminates the dispatch overhead of an interpreter, but it does nothing to eliminate the overhead of late binding (dynamic lookups) and dynamic type checks.

When LISP was originally created, it was not known how to make dynamic languages fast. Some simple implementation strategies were devised to improve performance, such as the use of type tags [18], which served to accelerate dynamic type tests. However, many crucial advances in the realm of program analysis had not yet been made.

As such, most believed that in order for dynamic languages to match the performance of statically typed languages, specialized hardware would be necessary.

In 1973, Richard Greenblatt and Thomas Knight of the MIT AI lab initiated the development of the MIT LISP Machine [1]. This was a computer with specialized hardware optimized to run LISP. This machine used a tagged architecture which implemented support for tag bits in hardware. That is, each word of memory contained a few additional bits to identify the kind of data stored (e.g. a `cons` pairs, pointers, small integers). The encoding of lists was compressed with CDR encoding, which allowed consecutive pairs to occupy only one word of memory and improved garbage collection performance. Various CPU instructions were also specialized for LISP programs.

In parallel with the development of the LISP machine, Xerox PARC began development of the Xerox Alto computer [34] and the Smalltalk programming language in 1972 [27]. This programming language featured dynamic typing and object-oriented polymorphism in the form of duck typing. The co-development of Smalltalk along with the Alto computer allowed engineers to take advantage of microcode to implement machine instructions optimized for the language, as was done in the LISP machine. In addition to this, new software tricks were invented to optimize specific language features, most notably inline caches [14], which greatly improved the performance of object property lookups.

The late 1970s saw important developments in program analysis. A seminal paper by Cousot & Cousot introduced the concept of abstract interpretation [13], a powerful technique for simulating the execution of a program at the symbolic level, which made it possible to prove properties about programs. Such techniques made it possible to analyze dynamically typed programs and eliminate some of the dynamic dispatch and dynamic typing overhead.

In the mid-1980s, the Self programming language, designed by David Ungar and Randall Smith at Xerox PARC, was the proving ground for several advances in the world of dynamic language optimization. Multiple techniques, including type feedback, polymorphic inline caches and path splitting were devised. These made it possible for Self to perform competitively on inexpensive commodity hardware.

The 1990s saw renewed interest in dynamic languages, with the advent of the web, and languages such as Perl, Python, Ruby, JavaScript and PHP. These languages were originally all interpreted, as fast personal computers made performance seem less relevant. However, as interactive websites became increasingly complex applications, both on the client and server side, there has been an increasing push to keep narrowing the performance gap between dynamic and static languages.

2.4 State of the Art JavaScript VMs

Early JS implementations were interpreted. This seemed sufficient then, because the language was largely used as glue code in otherwise static webpages. Over time, people became increasingly reliant on JS to create webpages that were no longer simply static, but rather dynamic, single-page applications. Facebook, Gmail and Google Docs are good examples of this. These web applications do not display static HTML files, but rather DOM tree nodes generated on the fly by JS code.

The appearance of dynamic web applications drove the need for faster JS implementations. Google took the lead by introducing the first version of its V8 JS engine in 2008, which featured a method-based JIT compiler. Mozilla followed shortly after with TraceMonkey, a tracing JIT compiler [16] for JS, in 2009. Today, all commercial JS implementations incorporate JIT compilers.

Since the beginning, web browser vendors competed for the JS performance crown. This led to the rediscovery of several dynamic language optimization techniques pioneered by Smalltalk and Self [4], such as type feedback, polymorphic inline caching, and the use of maps to optimize object property accesses. Classic optimization tricks such as boxing and tagging numerical values [18] were also borrowed from the LISP world. Even though the JS specification states that numerical values are all floating-point, both V8 and SpiderMonkey distinguish between integer and floating-point values internally, so that lower-latency integer machine instructions are used where possible.

Trace compilation initially gave TraceMonkey a performance advantage over other JS engines because of its fast compilation times, its capability for deep inlining, and

because it was possible to implement inexpensive type-specialization optimizations [17] in a tracing JIT. This was at a time when type analysis of JS programs was considered impractical and too expensive for use in a JIT compiler.

The competition in the JS performance arena has not ceased. After dynamic web applications, there has been a push for web browsers to become software platforms, with web applications completely replacing desktop applications. This drove the need for research into JS performance optimization, with the aim of getting the performance of JS as close as possible to that of programming languages used in the desktop realm, such as C, C++ and Java.

TraceMonkey was initially highly competitive, but it suffered from uneven performance. The code it produced was very competitive on benchmarks containing small, predictable loops, but was subpar elsewhere. It was retired in late 2011, in favor of a method-based JIT compiler. Today, commercial JS engines have converged towards multi-stage architectures, typically comprising an interpreter, a baseline (non-optimizing) JIT compiler and one or two levels of optimizing compilers.

Much of the JS code on any given webpage is run only once during initialization, or only run a few times. For such code, interpretation suffices, as it is typically faster than the time needed to generate machine code. The key idea behind multi-stage VM designs is that optimizations run within a JIT compiler must pay for themselves. That is, an optimization is only worth applying to a piece of code if the resulting reduction in execution time will be greater than the amount of time spent applying the optimization. The interpreter and baseline JIT allow for fast startup times, and the optimizing JITs apply expensive, heavyweight optimizations specifically to long-running code.

Until recently, type analysis of JS programs was considered highly problematic and difficult to achieve. However, a hybrid type analysis for JS has been developed by Mozilla and integrated into their SpiderMonkey JS engine in 2012 [19]. The analysis copes with the uncertainty introduced by constructs such as `eval` using speculative mechanisms and deoptimization [23]. Essentially, code is analyzed and optimized based on information available at the time the analysis is run. The optimized code may then be deoptimized later if speculative typing constraints turn out to be violated. This analysis

is relatively expensive, and so is only used at the highest optimization level.

Truffle/JS [38] is a JS engine created by Oracle which runs outside of a web browser, and is based on the Truffle/Graal framework [39]. A notable optimization published by the Truffle team is the encoding of JS object property types in object shapes (maps) [37] to allow their optimizing JIT to extract type information on property reads. Google V8 implements a similar optimization, but details of their implementation are unpublished.

2.4.1 Tricks of the Trade

This subsection lists many of the optimizations and tricks used by modern JS VMs to maximize performance.

Tag Bits. This is a kind of type tagging scheme [18]. The lowest bits of machine words are used to identify the kind of value being stored. Typical categories are integers, floating-point numbers, objects and strings.

NaN Tagging. Systems using tag bits typically box floating-point values via pointer indirection, which is inefficient. NaN tagging is a type tagging scheme in which bit patterns corresponding to floating-point NaN values are used to tag values which are not floating-point, but floating-point values themselves are untagged.

Small Integers. The ES5 specification states that all numbers must behave like IEEE double precision floating-point values. However, all modern JS VMs attempt to use machine integer operations when possible, because these have a lower latency than floating-point operations, but also to maximize ILP. As such, small signed integers, usually 31 or 32 bits in size, are used along with dynamic overflow checks to implement common arithmetic operations.

Lazy Method Compilation. Methods are only compiled to bytecode or IR when they are first executed. This avoids spending compilation time on code which never executes.

Lazy Parsing. When parsing a source file, the parser quickly verifies that the syntax of the entire file is valid, but it does not generate AST nodes for the inside of function bodies. The function bodies are only fully parsed when they are first executed.

Zone Allocation. Instead of allocating and deallocating AST and IR nodes individually, they are allocated by incrementing a pointer inside of a pre-allocated chunk of memory specific to the function being parsed or compiled (called a zone). Later, all memory allocated in the parsing and compilation of a function is freed at once, by discarding the zone. This decreases both the cost of allocation and deallocation of compiler data structures.

Staged Architectures. The VM has multiple stages of compilation and optimization. Usually an interpreter, a baseline JIT compiler, and an optimizing JIT compiler. Code that is run just once is interpreted, and only the longest running functions are compiled with the optimizing JIT compiler. Costly, advanced optimizations are applied selectively to code that is worth the compilation time expenditure. This idea dates back to the Self-93 implementation, which had both a baseline and an optimizing JIT compiler [20, 21].

Parallel Background Compilation. Functions deemed hot and worth optimizing are compiled using the optimizing JIT in one or more background threads. This allows execution to continue elsewhere while expensive optimizations are performed, thereby saving time and helping to avoid blocking the execution of the program.

Type Feedback. This idea dates back to Self [20]. It consists in using profiling methods or hooks into the runtime library to determine which types occur at run time and optimize code in accordance. V8 and SpiderMonkey use inline caches in their baseline JIT compilers to discover which types operators are applied to. Optimizing JITs can then inline code appropriate to the type combinations seen.

Inline Caching. The idea of inline caches dates back to implementation of the Smalltalk system [14]. The concept was later developed into that of Polymorphic Inline Caches (PICs) as part of the Self project [22]. A sequence of inline machine code instructions is made to test for specific types or object shapes and patch itself to execute the appropriate action given the input types seen. Inline caches are a way to optimize operations that could apply to many different types by specializing them on the fly to remove dynamic overhead.

Local Type Analysis. Intraprocedural type analysis is used to eliminate dynamic type checks inside of a method. This is only done in optimizing JITs, after selected callees are inlined so as to expose more type checks.

Speculative Optimization. V8 performs a number of optimizations speculatively. For instance, it speculates that integer overflows will not occur, and uses this to improve local type analysis results. When the speculation turns out to be incorrect, the function in which this occurs is deoptimized, that is, the speculatively optimized code is discarded.

Object Shapes. Object shapes (sometimes called "hidden classes") derive from the concept of maps, which originated in Self [4]. They are structures which encode the memory layout of one or more objects. Each object has an associated shape which tells us which properties are in the object and the memory offset at which each property is stored. Shapes can also be used to associate metadata with each property.

Loop Unrolling. Optimizing JITs in modern JS VMs perform loop unrolling to maximize performance.

Auto-Vectorization. Some operations which can make use of vectorization are automatically detected through analysis, and optimized using SIMD vector instructions.

Allocation Sinking. Allocation sinking [28] is an optimization which tries to delay the allocation of an object for as long as possible, in the hopes that this allocation will be found unnecessary. This optimization tends to pay off when inlining deeply. It can be used to eliminate the allocation of closures.

Ropes. JS strings are immutable, and so string concatenation operations implicitly involve copying both of the strings being concatenated into a new string object. This is problematic because many JS developers will generate strings by repeatedly appending new content to a string in a loop, and the execution time for this is quadratic in the size of the output. The solution to this issue, in languages such as Java, has been to force programmers to use a special mutable string representation (the `StringBuffer` class). Ropes are a string representation (hidden from the programmer's view) which is used to concatenate strings lazily. Instead of immediately concatenating two strings, a rope object will be allocated to indicate that a concatenation should be performed. The rope is only materialized into a real string object when some operation tries to access the

underlying data. With this system, data can be repeatedly appended to a string in linear time.

Dependent Strings. Dependent strings are another string representation which is meant to optimize the performance of manipulations on immutable strings. In particular, they are meant to optimize string slicing operations. When requesting a substring, instead of copying the characters in the specified range, a dependent string object is allocated, which stores a reference to the original string along with the index and length of the range. This makes it possible to defer the copy operation, which can improve performance if the substring is later appended to some other string, or if the result is simply never used.

2.5 Summary

Dynamic languages are not a new invention, they date back all the way to the late 1950s, with the invention of LISP. However, in the early days, it was not known how to make them run fast on general-purpose computer hardware. For a long time, dynamic languages were at a significant disadvantage compared to statically typed languages due to the run-time overhead incurred by late binding and dynamic typing. The advent of Python, JavaScript and other dynamic languages which powered the internet revolution sprouted new interest in making dynamic languages perform competitively.

The following chapters will focus on our own research and exploration of a JIT compiler architecture designed with dynamic optimization in mind, and in particular, our efforts to find ways to effectively eliminate dynamic type checks without incurring a heavy compilation time cost.

CHAPTER 3

A SELF-HOSTED JAVASCRIPT VM

The article presented in this chapter details the implementation of Tachyon, a self-hosted VM for JavaScript, itself written in JavaScript. This implementation was built from scratch, without using code generation frameworks such as LLVM [29, 30]. The register allocation, code generation and even the assembler were written in JavaScript.

Tachyon was built as a platform to explore the implementation of new program analyses and optimizations for dynamic languages, JavaScript in particular. Self-hosting, writing Tachyon itself in JavaScript, was seen as a way to simplify the implementation of the compiler, and as a way to test the capabilities of the system.

The article presented in this chapter was submitted and accepted at the Dynamic Languages Symposium (DLS) 2011, and then published in SIGPLAN Notices [10]. Tachyon took two graduate students approximately two years to build and weighs in at approximately 75000 lines of code. Its source code is open source and available on GitHub [1].

3.1 Design of Tachyon

Tachyon features a custom SSA-based IR inspired from that of LLVM [30]. As with LLVM, the Tachyon IR is structured into single-entry single-exit basic blocks terminated with branching instructions. This IR has built-in support for dynamic typing. That is, each SSA value has an associated type tag, and there are low-level type test primitives to determine what the type tag of a given value is at run time. There are also low-level primitives which represent low-level machine instructions for integer and floating-point arithmetic, as well as memory accesses (loads and stores).

One of the most important distinguishing features of Tachyon is that each JS operator was implemented in terms of calls to runtime library functions, themselves written in an extended dialect of JS. Implementing these runtime functions requires access to low-

1. https://github.com/Tachyon-Team/Tachyon

level type test and machine instruction primitives. In Tachyon, this is done with what we term "inline IR", which exposes these primitives as if they were callable functions in our extended JS dialect, similarly to the way C compilers allow inline assembly.

The way Tachyon implements JS operators as runtime library functions is in opposition to the design of mainstream JS compilers. These typically implement higher level IRs, where each JS operator is a polymorphic IR instruction able to operate on multiple operand types. For instance, in V8 and SpiderMonkey, the JS addition operator is represented by an `add` IR instruction which can operate on integers, floating-point values, strings, objects, etc. The machine code generated for each instance of this instruction is specialized based on available type information.

In Tachyon, we instead represent the JS addition operator as a function call which can be inlined, and then type-specialized like any other function. This simplifies the implementation of the JIT compiler, because the compiler does not need special code to know how to effectively specialize machine code for each primitive operator. It also eliminates a number of corner cases which are problematic to deal with.

For instance, consider that the JS addition operator can operate on values of any type, including objects. If we were to "add" a string to an object, the `toString` method of the object would be called. This would then mean that any addition operator must internally be able to encode the logic of a JS function call. This function call could in turn throw an exception, and such corner cases must be handled correctly. As such, writing a function which directly generates machine code for JS operators can be very complicated. This is our motivation in building an IR which encodes lower level instructions instead. The result is that instructions in Tachyon's IR are more "atomic" and predictable in nature, and the compiler is able to analyze and optimize the behavior of JS primitives as it would any other JS function.

3.2 From Tachyon to Higgs

Tachyon is able to run several benchmarks from the SunSpider suite and successfully compiles itself to x86 machine code. We chose, however, not to pursue its development

any further, because of issues with performance, and the amount of development effort required to keep developing Tachyon to the point where it would be suitable for implementing a more sophisticated JIT compiler.

One major issue with the design of Tachyon, is that it was running on top of the V8 VM. As such, in order for programs compiled by Tachyon to communicate with Tachyon itself, these programs would need to call back into Tachyon functions. This required exposing callbacks to JS functions running on V8 (written in C++) to machine code we had generated ourself. Doing so would be technically challenging, but also guaranteed to perform fairly poorly.

The Tachyon VM is able to compile itself to x86 machine code in RAM. It is, however, mostly designed in the same way as a static, ahead of time compiler. That is, it compiles a series of source files all at once, and the way it compiles this code is not affected by the program's state. Tachyon does not, for instance, implement profiling or lazy compilation of methods, as this would require compiled programs to call back into the JIT.

These limitations are what motivated us to begin work on Higgs, a successor to Tachyon written in D. The D programming language was chosen because it is a systems programming language, with support for things such as pointer manipulation and specifying the calling convention of functions, which also offers the niceties of garbage collection and some basic type inference.

Higgs is a direct descendant of Tachyon. It retained Tachyon's self-hosted implementation of the JS runtime and standard library, its large bank of unit tests, as well as several crucial aspects of its design, such as the SSA-based IR. Most importantly, BBV relies on low-level type test instructions being exposed in the IR, as was done in Tachyon.

Bootstrapping a Self-Hosted Research Virtual Machine for JavaScript

An Experience Report

Maxime Chevalier-Boisvert
Université de Montréal

Erick Lavoie
Université de Montréal

Marc Feeley
Université de Montréal

Bruno Dufour
Université de Montréal

ABSTRACT

JavaScript is one of the most widely used dynamic languages. The performance of existing JavaScript VMs, however, is lower than that of VMs for static languages. There is a need for a research VM to easily explore new implementation approaches. This paper presents the Tachyon JavaScript VM which was designed to be flexible and to allow experimenting with new approaches for the execution of JavaScript. The Tachyon VM is itself implemented in JavaScript and currently supports a subset of the full language that is sufficient to bootstrap itself. The paper discusses the architecture of the system and in particular the bootstrapping of a self-hosted VM. Preliminary performance results indicate that our VM, with few optimizations, can already execute code faster than a commercial JavaScript interpreter on some benchmarks.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, code generation, run-time environments*

General Terms

Algorithms, Performance, Design, Languages

Keywords

JavaScript, virtual machine, compiler, self-hosted, optimization, implementation, framework

1. INTRODUCTION

JavaScript (JS) [8] was designed by Netscape in 1995 for client-side scripting of web pages. Since then, all mainstream web browsers have integrated a JS Virtual Machine (VM) which can access the Document Object Model, placing JS in a unique position for implementing web applications.

DLS'11, October 24, 2011, Portland, Oregon, USA.

Due to this, JS is currently one of the most widely used dynamic programming languages. Each of the main browser vendors has built their own VM, some being open-source (Apple WebKit's SquirrelFish[1], Google Chrome's V8[2], Mozilla Firefox's SpiderMonkey [6]) and some closed-source (Microsoft Internet Explorer 9's Chakra[3]), all of which use Just-In-Time (JIT) compilation.

With the increased use of JS in web applications, the performance of a browser's JS VM has gained in importance in the past few years, even appearing prominently in the browser's marketing literature. However, the performance of the best JS VMs is still lower than the performance of VMs for static languages. Although there is clearly a need to design more efficient VMs, there has been relatively little academic research on the implementation of JS. We believe this is mainly due to the lack of easily modifiable JS tools, and in particular a research VM, which would allow easy experimentation with a wide variety of new approaches. The mainstream VMs, even if they are open-source, are not appropriate for this purpose because they are large systems in which it is tedious to change even conceptually simple things (such as the function calling convention, the object representation, the memory manager, etc.) without breaking obscure parts of the system. This has motivated us to begin the design of a family of JS VMs and tools suitable for research and experimentation.

This paper is an experience report on the design of *Tachyon*, our first JS VM. Tachyon is a work in progress, and this paper discusses the current state of the system. Specifically, Tachyon has now reached the point where it bootstraps itself. The discussion of the bootstrap process is thus a central aspect of this paper.

1.1 Self-Hosting

The most remarkable feature of Tachyon is that it is written almost entirely in JS. We expect this self-hosting to yield some important benefits compared to a VM written in another language. It is typical to write JIT-based VMs in a low-level host language (e.g. C/C++) in order to have fast JIT compilation. But this is at odds with the productivity advantage of a high-level language, such as JS, for writing the complex algorithms that are found in compilers. This is

[1] http://trac.webkit.org/wiki/SquirrelFish
[2] http://code.google.com/p/v8/
[3] http://en.wikipedia.org/wiki/Chakra_(JScript_engine)

an important issue for a research compiler where rapid prototyping of new compilation approaches is desirable. Since we expect Tachyon's code generation to eventually be competitive with compilers for static languages, we also believe the self-hosting will not cause the JIT compiler to be slow. Self-hosting also has the advantage of a single runtime system (memory manager, I/O, etc), which eliminates code duplication as well as conflictual interactions of independent client and host runtime systems.

1.2 JavaScript

Tachyon aims to implement JS as specified in the ECMA-Script 5 (ES5) [8] specification. Although the infix syntax of JS superficially resembles that of Java, the JS semantics have much more in common with Python and Smalltalk. It is a dynamic language, imperative but with a strong functional component, and a prototype-based object system similar to that of SELF [3].

A JS object contains a set of *properties* (a.k.a. *fields* in other OO languages), and a link to a parent object, known as the object's *prototype*. Properties are accessed with the notation *obj*.*prop*, or equivalently *obj*`["`*prop*`"]`. This allows objects to be treated as dictionaries whose keys are strings, or as one dimensional arrays (a numeric index is automatically converted to a string). When fetching a property that is not contained in the object, the property is searched in the object's prototype recursively. When storing a property that is not found in the object, the property is added to the object, even if it exists in the object's prototype chain. Properties can also be removed from an object using the `delete` operator. JS treats global variables, including the top-level functions, as properties of the *global* object, which is a normal JS object.

Anonymous functions and nested functions are supported by JS. Function objects are closures which capture the variables of the enclosing functions. Common higher-order functions are predefined. For example, the `map`, `forEach` and `filter` functions allow processing the elements of an array of data using closures, similarly to other functional languages. All functions accept any number of actual parameters. The actual parameters are passed in an array, which is accessible as the `arguments` local variable. The formal parameters in the function declaration are nothing more than aliases to the corresponding elements at the beginning of the array. Formal parameters with no corresponding element in the array are bound to a specific `undefined` value.

JS also has reflective capabilities (enumerating the properties of an object, testing the existence of a property, etc) and dynamic code execution (`eval` of a string of JS code).

The next version of the standard is expected to add proper tail calls, rest parameters, block-scoped variables, modules, and many other features. Even though we are currently targeting ES5, we have been careful to keep Tachyon's design amenable to implementing the expected additions without extensive refactoring.

1.3 Contributions

This paper presents Tachyon, a meta-circular JS VM. It aims to serve as a case study for the design of a meta-circular VM for dynamic languages. The main contributions of this paper are:

- A presentation of the design of our compilation pipeline (Section 3).
- The design of low-level extensions to JS for manipulation of memory, compatible with the existing syntax (Section 3.5).
- An execution model for the VM (Section 4.1).
- A description of the bootstrap process required to initialize the VM given the execution model (Section 5).

Note that for the remainder of the paper, we use the term *compiler* to designate subsystem responsible for translating JS code to native code, and the term *VM* to refer to the combination of the runtime system and the compiler.

2. RELATED WORK

The literature on the design of meta-circular VMs for dynamic languages is rather sparse. To the best of our knowledge, no comprehensive synthesis of issues and opportunities has been done. There are, however, documented examples for some languages.

Squeak is a recent implementation of Smalltalk, written in Smalltalk. The Squeak VM is a bytecode interpreter written in a restricted, non-object-oriented subset of Smalltalk [7] that can be easily compiled to C code. This approach prevents usage of more expressive features of the language for the implementation of the VM. In contrast, Tachyon can use the entire JS subset it supports in its own implementation.

JikesRVM was the first meta-circular VM to show that high-performance was compatible with meta-circularity. Tachyon uses a similar mechanism to the JikesRVM *magic* class to expose primitive operations (Section 3.5).

Klein, a meta-circular implementation for Self [18], showed that mirror-based reflection [2] could foster much code reuse. Tachyon provides the reflection mechanisms specified in ES5 but those do not comprise mirror-equivalent mechanisms, preventing usage of the Klein implementation techniques for serializing objects in a binary image and remote debugging. Tachyon creates the objects needed at run-time during the initialization phase instead of relying on mirrors to access already existing objects (Section 4.2). Tachyon uses an x86 assembler that we implemented in JS but unlike Klein's assembler, it is not self-checking.

PyPy [16] uses a rather different approach from other meta-circular VM projects. It does not directly compile Python code into executable code. Rather, their approach involves describing the semantics of a bytecode interpreter for a programming language (e.g. Python), and generating a virtual machine (e.g. generating C code) that supports the described language. It is able to improve on the performance of the raw bytecode interpreter by applying . The system now supports most of Python and significantly improves upon the performance of the stock CPython distribution, partly due to a tracing JIT compiler [1]. Tachyon is hand-coded, and does not use automatic generation techniques.

In contrast to the aforementioned systems, Tachyon does not use a bytecode representation for compiled code. It compiles JS directly to native code. A low-level Intermediate Representation (LIR) based on a Static Single Assignment (SSA) form is used as a platform-neutral representation for compiled code (Section 3.3) instead of bytecode.

Other researchers have studied the compilation of JS. Loitsch has proposed an approach to leverage the similarity between JS and Scheme by using it as a target lan-

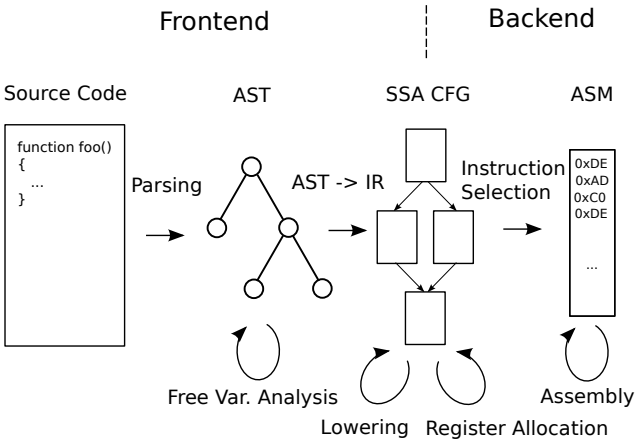

Figure 1: Overview of the compiler phases and representations.

guage for a JS compiler [11]. Gal *et. al.* have described TraceMonkey [6], a trace-based JS compiler that achieves significant speedup by exploiting type stability in (possibly nested) hot loops to perform type specialization and other optimizations. Experiments on a popular benchmark suite reveal that the technique can achieve speedups up to 25x.

The TraceMonkey implementation was later extended by Sol *et. al.* [17] to further remove provably unnecessary overflow tests at run-time using a flow-sensitive range analysis. The authors show that their technique is able to remove more than half of the overflow checks in a sample of real JS code from top-ranked web sites. Logozzo and Venter [10] have also proposed an analysis that determines ranges as well as types (e.g. 64-bit floating-point or 32-bit integer) of numerical variables. The authors report up to 7.7x speedup for some numerical benchmarks. Such techniques could be implemented as part of Tachyon.

Some empirical studies can also shed some light on the behavior of real-world JS applications, for example by comparing properties of standard benchmarks with real applications [15, 13], or by studying the use of particular language features such as *eval* [14]. These studies will help direct future efforts and implementation choices in our project.

3. COMPILER

The compiler operates on a succession of intermediate representations getting progressively closer to native assembly code. Four representations are used with the bulk of the work being done on the third one:

- Source code: string representation of the JS code
- Abstract Syntax Tree (AST): tree representation of declarations, statements and expressions
- Intermediate Representation (IR): Control Flow Graph (CFG) in SSA form. Used to represent both the High-Level IR (HIR) and Low-level IR (LIR)
- Assembly code (ASM): array of bytes and associated meta-information representing the encoded assembly instructions

A number of phases either produce those representations or transform them. The front-end comprises the platform independent phases:

- Parsing and free variable analysis: the source code is translated into an AST and free variables are tagged (Section 3.2).
- AST->IR: the AST is transformed into the HIR (Section 3.3).
- Lowering: the HIR is transformed into the LIR and optimizations are performed (Section 3.4).

The back-end comprises the platform dependent phases:

- Register allocation: operands and destination temporaries are associated to physical registers or memory locations in the LIR instructions (Section 3.7).
- Instruction selection: the LIR after register allocation is transformed directly to a partial encoding of ASM (Section 3.10).
- Assembly: the final encoding for ASM is generated (Section 3.11).

Both representations and phases are illustrated in Figure 1.

3.1 Supported JS Subset

Tachyon currently supports enough of ES5 to bootstrap itself. The VM supports strings, arrays, closures, constructors, objects, prototypes and variable argument count. The Boolean, Error, Function, Math, Number, Object, String objects and a subset of their methods from the standard library are also supported. The internal representation for numbers uses fixed precision integers. Bitwise, logical and arithmetic operations are supported.

These functionalities were sufficient to implement the data structures and algorithms needed in pure JS. For example, we required associative tables, sets, infinite precision integers, graphs and linked lists. The VM does not yet support regular expressions, floating-point numbers[4], exceptions, object property attributes (read-only, enumerable, etc.), getters and setters.

While Tachyon does not currently support floating-point numbers, this does not confer it an unfair performance advantage over other JavaScript VMs that do. Type and overflow tests are already inserted in the generated code so as to handle floating-point operations once they are implemented. Hence, the performance is the same as if these operations were supported, but Tachyon cannot yet run code that would invoke floating-point operations.

3.2 Parsing and AST

The parser is automatically generated from the WebKit *Yacc* JS grammar. A script transforms the grammar into S-expressions which are fed to a Scheme LALR parser generator, and the resulting parsing tables are pretty printed as JS arrays. A hand written scanner and a LALR parser driver complete the parser.

[4]At the time of publication, regular expressions and floating-point numbers are implemented but not fully tested, and are therefore excluded from the supported JS subset.

The AST is then traversed to compute some properties (scope of variables, set of free variables, set of variables declared in a function, use of `eval` and `arguments` variables within a function, etc). The AST is also transformed to

- add debugging code when this is requested (for example tracing function entry and return),
- rewrite some constructions into a canonical form, such as transforming *obj*`.`*prop* into *obj*`["`*prop*`"]`, and transforming variable declarations into assignments,
- rewrite formal parameters accesses to indexing of the `arguments` object, when the `arguments` variable is accessed in the function.

3.3 Intermediate Representation

After the AST transformations, the AST is translated into the IR. This IR is in SSA form [5]. It comprises low-level type annotations which were inspired by those of LLVM [9]. We have chosen an SSA-based IR because such an IR is closer to machine code and we expect it will be efficient for recompilations to be done directly on the IR without having to restart the compilation at the AST level. The Tachyon IR is loosely divided into HIR and LIR layers, which are both refinements of a generic SSA-based IR. ASTs are initially translated into the HIR, and the HIR code is later translated into the LIR.

Each function compiled by Tachyon has an associated CFG divided into basic blocks. At the beginning of each basic block is a possibly empty list of phi nodes implementing parallel conditional assignments. These phi nodes are followed by a series of instructions, each of which produces zero or one output value. Branch instructions are only allowed at the end of basic blocks. These can either be return instructions, direct jumps, if-test instructions with two possible targets, throw instructions, or call instructions. Call instructions can have a regular continuation target and an exception target, making exception control-flow explicit.

The HIR comprises high-level operations meant to represent dynamically-typed JS operators directly. It comprises add and subtract primitives, for example, which represent the JS + and – operators, and can thus operate on any JS type. The HIR is easy to analyze in terms of JS semantics, but remains abstract. It is eventually transformed into the LIR, which translates fairly directly to machine code.

The LIR is meant to closely represent the kinds of operations that most modern computer processors implement. It is more verbose but also more expressive. By controlling how specific instances of HIR constructs are translated into LIR based on type information, the code can be specialized and optimized. The LIR exposes some low-level types such as pointers, garbage-collected references and machine integers. It is close to the expressiveness level of C code, and comprises instructions for native integer, floating-point and pointer arithmetic.

We have chosen to express these low-level operations in the IR so as to allow further optimizations on the LIR to be implemented in a portable way in the front-end. Despite being closer to machine code, the LIR remains relatively machine-agnostic. It is in SSA form and does not expose highly machine-specific details such as machine registers, stack frame formats and calling conventions. These are to be specified by a back-end tailored to the target architecture.

3.4 Optimizations

The front-end currently implements several commonplace optimizations which can operate on both the HIR and LIR. These include function inlining, Sparse Conditional Constant Propagation (SCCP) [19], Common Subexpression Elimination (CSE) as well as dead code elimination, strength reduction and peephole optimization patterns. The optimization patterns simplify control flow graphs by eliminating known patterns of redundant phi nodes, branches and instructions.

The optimizations we have implemented are fairly basic. They help improve the quality of the code generated by Tachyon, but do not yet attack the more critical performance issues involved in optimizing JS code. In particular, they are unable to optimize for more likely code paths, optimize based on type information, or reduce the cost of property accesses. We aim to implement more advanced analyses and optimizations in the future, such as optimistic optimization techniques and inlining (see Section 7).

3.5 JS Extensions

The JS language lacks some of the low-level functionality required to implement a JIT compiler. More specifically, it does not allow accessing raw memory directly, nor does it allow the execution of arbitrary code, or even file system access. As such, Tachyon is not written in pure JS, but instead in an extended dialect of the language.

One obvious way to extend JS was to implement a Foreign Function Interface (FFI) to allow Tachyon to call C functions. Once such an interface is in place, JS code can be made to call C code to implement the functionality that JS itself does not provide. One possible design choice would have been to make no further extensions to JavaScript and implement all the missing functionality required to write a JIT compiler in C. However, this would imply the implementation of significant portions of the compiler in C directly, which could be both difficult to maintain and problematic for performance, as FFI function calls are costly, opaque and non-inlinable.

Since Tachyon compiles JS code to a low-level IR that has an expressive power similar to that of C, we have been able to further extend JS with typed variables directly representing pointers and machine integers. By default, all variables in Tachyon have a *boxed* dynamic type, which can store any JS value, but functions can be annotated to say that they take typed variables as arguments (e.g.: 32 bit integer, raw pointer). External C primitives that take typed variables as input and return typed values can also be called.

The C primitives exposed to Tachyon only implement low-level operations (e.g.: `malloc/free`, file and console I/O, timing functions). To implement Tachyon primitives needing direct access to memory or lower-level hardware capabilities, we directly expose LIR instructions to JS code, in *Inline IR* (IIR), a concept similar to inline assembly. Compared to inline assembly, however, Inline IR code is more machine-agnostic and more readable. It makes it possible, for example, to call the LIR load and store instructions as if they were JS functions when one needs to read or write to memory.

The snippet of code below shows the `readConsole` C function being registered with Tachyon:

```
regFFI(new CFunction(
   'readConsole',
   [new CStringAsBox()],
   new CStringAsBox(),
   params
));
```

The function `readConsole` takes a prompt string as argument and reads a user-input string from the console, which is returned. The snippet of code shown creates a proxy function that will convert a boxed string argument (JS string) into a `char*` string pointer for the C function, and perform the reverse conversion on its return value. The `readConsole` function is then used in Tachyon's read-eval-print loop as if it were an ordinary JS function.

For the inline IR, we have designed a syntax that can be parsed by an unmodified JS parser. The example in Figure 2 shows a function used to convert C ASCII strings into JS strings usable by Tachyon. This function uses the `iir.load` instruction to read character values directly from the C string, and casts them into 16-bit UTF-16 characters using the `iir.icast` instruction. The `iir.load` instruction is annotated to say that the value it is loading is a single byte (`i8` type). The output value of the instruction is thus not a boxed value, but a typed LIR value.

```
function cStringToBox(strPtr)
{
    "tachyon:static";          // Statically-linked func.
    "tachyon:noglobal";        // No global object access
    "tachyon:arg strPtr rptr"; // strPtr is a raw pointer

    // If the string pointer is NULL, return the JS null
    if (strPtr === NULL_PTR)
        return null;

    // Compute the string length
    for (var strLen = pint(0); ; strLen++)
    {
        var ch = iir.load(IRType.i8, strPtr, strLen);
        if (ch === i8(0))
            break;
    }

    // Allocate a string object
    var strObj = alloc_str(strLen);

    // For each character
    for (var i = pint(0); i < strLen; i++)
    {
        var cCh = iir.load(IRType.i8, strPtr, i);
        var ch = iir.icast(IRType.u16, cCh);
        set_str_data(strObj, i, ch);
    }

    // Compute the hash code for the new string
    compStrHash(strObj);

    // Attempt to find the string in the string table
    return getTableStr(strObj);
}
```

Figure 2: Function using Inline IR (IIR).

3.6 Implementation of the Primitives

Tachyon implements the JS semantics by mapping the primitive operations of the JS language to HIR instructions which can be translated into one or more LIR instructions. This translation is currently done in a very straightforward way: each HIR instruction maps to a call to a primitive function, which may or may not be inlined as the HIR is translated into LIR. The primitive functions may or may not use inline IR functionality to implement their semantics. The `add` primitive, for example, makes use of an LIR instruction implementing a machine addition with an overflow check so that integer additions can be optimized.

The primitive functions `getProp`, `putProp` and `hasProp` implement object property access. Other primitives implement basic operations on string and array objects. Because it would be tedious to constantly have to perform pointer arithmetic and invoke LIR instructions to access the memory layouts of these objects, we have chosen to take a helpful shortcut. Memory layouts are described in Tachyon using layout objects, which store mappings of field types and names to memory offsets. These are similar to C structs. We then use metaprogramming to auto-generate extended JS code (inlinable setter and getter methods) to access each field of a given layout. This greatly improves the readability and maintainability of our runtime implementation.

It is possible to use extended JS anywhere inside the Tachyon code, however, we have made a conscious effort to try and limit its use to the implementation of low-level primitives. Tachyon's implementation of the JS standard library, as well as most of Tachyon's implementation remains mostly written in pure JS code. This gives us the added benefit that the Tachyon LIR can be changed without an enormous refactoring effort being required.

3.7 Register Allocation

The most important compromise to be made when designing a register allocation algorithm is between compilation speed and quality of the resulting code, namely maximizing usage of registers for the most frequently executed instructions and minimizing the number of moves between registers and memory. Although earlier work was biased toward the latter, recent algorithms targeted at JIT compilers now give more weight to the former.

We initially chose to implement the Linear Scan (LS) register allocator [12], specialized for the SSA representation [20] as it seemed well-suited to our choice of IR and was reported to produce code competitive with a graph-coloring algorithm. However, in improving the implementation, we remarked that operating on live intervals instead of directly on the CFG makes the task of modelling the target architecture constraints more difficult. This motivated the implementation of a second, On The Fly (OTF) allocator, also operating on the SSA representation. Both algorithms use the Most Distantly Used heuristic for spilling.

Although a detailed analysis has yet to be performed, initial tests on simple benchmarks suggest that our OTF implementation is simpler, faster and produces code of similar or better quality than our LS algorithm. The current implementation of Tachyon can use either interchangeably.

3.8 Calling Convention and Register Usage

To our knowledge, no systematic exploration of the design space has been done concerning calling conventions and register usage for compiled dynamic languages. From the very beginning, we anticipated using Tachyon to identify the most promising ideas. To accommodate such flexibility, the number and the nature of registers available for register allocation as well as for passing arguments to functions can be varied by modifying a single configuration object. We chose to reserve for a future time the factorization necessary for other possibilities such as using callee-save registers and

using a register to pass the return address.

Exploration of the different possibilities is planned for the near future. Currently, Tachyon uses a single configuration corresponding to our educated guess of what would be faster. Moreover, the same register usage and calling convention are used on both x86 and x86-64. Those choices are motivated by the desire to obtain a working compiler faster with the intention of revisiting the choices made in the future if they become a bottleneck for the execution speed.

The current calling convention uses four registers to pass arguments to a function, respectively for passing the closure pointer, the `this` object and the first two parameters of the function. The stack is used to pass the return address and the remaining parameters. A location on the context object (see Section 4.1) is used for passing the argument count. For performance reasons, direct support for C calls is implemented both for x86 and x86-64. Stack alignment and proper argument passing is inlined in the generated code.

The current register usage on x86 reserves five registers for register allocation, and three registers respectively for keeping a reference to the context object, the stack pointer and a scratch register. The latter is used as a temporary measure to lessen the constraints on register allocation but we intend to eliminate it eventually.

3.9 Function arguments handling

The flexibility gained by having variadic functions by default incurs a run-time overhead when the caller does not know the expected number of parameters for a variadic function. The current implementation uses an assembly language handler prepended at the entry point of each function. This handler manipulates the call stack to match the number of expected parameters. Extra arguments are removed and missing arguments are initialized to the *undefined* value.

As a special case, when the compiler generates calls to primitive functions, since the number of arguments and the non-usage of the `arguments` object are known at compile-time, a fast entry point is used which avoids passing and checking the argument count at run-time.

The `arguments` object also incurs a run-time cost because the object in question needs to be constructed. For functions using it, all arguments passed on the stack are copied in an array, before the check for the number of arguments is performed and the stack frame is manipulated. A handler for creating the `arguments` object is prepended to the variadic function handler for functions making use of the feature.

Note that the `arguments` object contains function arguments, but also a reference to the callee function object. This information is available to functions because the function object is a hidden argument in our default calling convention. Some JavaScript implementations also include a reference to the caller function in the `arguments` object. Tachyon does not provide this because it is not part of the ES5 standard. We do not believe it would be difficult or costly to implement, however.

3.10 Instruction Selection

Instruction selection is done after register allocation. It tries to exploit faster and/or smaller instructions when possible. For example, when performing an addition with an immediate value of 1, the `inc` instruction will be used instead of the regular `add` instruction. Most of the work concerns ensuring the proper location of operands with regard to x86 operand conventions, such as not having two operands in memory and having one of the operands being the destination of the instruction.

Instruction selection notation uses regular JS mixed with a selected subset designed to mimic the GNU assembler syntax. Inside regular code, an assembly language context is initiated by referring to the assembler object, `asm` by convention. Then, using `$` as an immediate value constructor, `mem` as a memory address constructor, variable names to refer to register objects, as well as cascading function calls [4], namely methods returning the `this` object, allows to write assembly code in the style shown below:

```
asm.
mov(eax, ebx).          // 000000 89 c3          movl %eax,%ebx
add($(1), ebx).         // 000002 83 c3 01       addl $1,%ebx
add(mem(0,esp), ebx). // 000005 67 03 1c 24    addl (%esp),%ebx
ret();                  // 000009 c3             ret
```

This has proved helpful in working at different levels of abstraction, all within the same language. For example, JS can be used as a metaprogramming language for writing assembly code by defining cascading functions implementing common idioms. The following example illustrates copying values from the stack to an array, using a for loop generator pattern:

```
var i = eax;

asm.
forLoop(i, ">=", $(0), function () // for (;i >= 0; --i)
{
    this.
    mov(mem(0, sp, i), temp).      // temp = sp[i]
    mov(temp, mem(0, arr, i));     // arr[i] = temp
}).
ret();
```

Interleaving assembly instructions with generator patterns makes assembly writing more convenient and arguably easier to read than with a regular assembler.

3.11 Assembly

Once a partial encoding for instructions has been generated by the instruction selection phase, a fixpoint on the assembly code generated is performed to find the minimal length encoding for the given assembly language instructions. This is necessary because the encoding length for some instructions varies as a function of the value of some operands. For example, encoding a relative jump with an 8-bit displacement uses only two bytes instead of five for a 32-bit displacement.

3.12 Compilation Example

```
function add1(n)
{
   return n + 1;
}
```

Above is the source code for a simple JS function which computes `n+1`. Due to the generic nature of the `+` operator, this either adds one if `n` is a number, or converts `n` to a string, if it isn't already one, and concatenates the string `"1"` to it.

Figure 3 illustrates the AST produced by our parser for the `add1` function. This AST includes the function declaration, the block of statements inside the function body, the variables and their scope, as well as the operator expression adding `n` to `1`, wrapped inside a return statement. The HIR produced for this function is shown below:

```
Program                          ("ex.js"@1.1-1.36:)
|-var= add1 [global]             ("ex.js"@1.1-1.36:)
|-func= add1 [global]            ("ex.js"@1.1-1.36:)
|-block=
| BlockStatement                 ("ex.js"@1.1-1.36:)
| |-statements=
| | FunctionDeclaration          ("ex.js"@1.1-1.35:)
| | |-id= add1 [global]          ("ex.js"@1.1-1.36:)
| | |-funct=
| | | FunctionExpr               ("ex.js"@1.1-1.35:)
| | | |-param= n                 ("ex.js"@1.15-1.16:)
| | | |-var= n [local]           ("ex.js"@1.1-1.35:)
| | | |-body=
| | | | ReturnStatement          ("ex.js"@1.20-1.33:)
| | | | |-expr=
| | | | | OpExpr                 ("ex.js"@1.27-1.32:)
| | | | | |-op= "x + y"
| | | | | |-exprs=
| | | | | | Ref                  ("ex.js"@1.27-1.28:)
| | | | | | |-id= n [local]      ("ex.js"@1.1-1.35:)
| | | | | | Literal              ("ex.js"@1.31-1.32:)
| | | | | | |-value= 1
```

Figure 3: AST for the `add1` function.

```
entry:
box n = arg 2;
box $t_4 = call <fn "add">, undef, undef, n, box:1;
ret $t_4;
```

As can be seen, this representation is rather concise and abstract. The value of the argument `n` is assigned to an SSA temporary. The primitive `add` function is then called to implement the behavior of the `+` operator applied to `n` and `1`. The result of this call is then returned. The two `undef` arguments to the call represent the closure and `this` object references, which are undefined in the case of primitive calls.

```
entry:
box n = arg 2;
pint $t_4 = and_box_pint n, pint:3;
if $t_4 === pint:0 then cmp_true else if_false;

cmp_true:
box $t_14 = add_ovf n, box:1 normal call_res overflow ovf;

call_res:
box phires = phi [$t_11 ovf], [$t_19 if_false], [$t_14 cmp_true];
ret phires;

ovf:
ref $t_9 = get_ctx;
box global_2 = load_box $t_9, pint:36;
box $t_11 = call <fn "addOverflow">, undef, global_2, n, box:1;
jump call_res;

if_false:
ref $t_17 = get_ctx;
box global_3 = load_box $t_17, pint:36;
box $t_19 = call <fn "addGeneral">, undef, global_3, n, box:1;
jump call_res;
```

Figure 4: LIR for the `add1` function.

The LIR for `add1` is produced by inlining the call to the primitive `add` function (see Figure 4). This results in many basic blocks being added to `add1`. This code implements the multiple semantics of the `+` operator. The tag bits of the operand values are first tested to see if both operands are integers. The test of the tag bits of the constant are eliminated by constant propagation. If `n` is an integer, we use the `add_ovf` instruction to perform an integer add with an overflow check directly on the bits of the values.

If the result overflows, the `add_ovf` instruction will branch to the `ovf` basic block, in which the `addOverflow` function is called to handle this case. If `n` was not an integer to begin with, the `addGeneral` function is called to implement the generic addition semantics, which may result in a string concatenation, for example. In all cases, the control-flow eventually reaches the `call_res` block and the final result of the addition is passed to the `phires` phi node, whose value is then returned.

```
0000                        <fn:add1>
/* stack adjustment prelude removed */
0051                        entry:
0051 89 c7                  movl %eax,%edi
0053 83 e7 03               andl $3,%edi
0056 85 ff                  testl %edi,%edi
0058 75 3b                  jne if_false
005a eb 00                  jmp cmp_true
005c
005c                        cmp_true:
005c 89 c7                  movl %eax,%edi
005e 83 c7 04               addl $4,%edi
0061 71 02                  jno call_res
0063 eb 08                  jmp ovf
0065
0065                        call_res:
0065 83 c4 04               addl $4,%esp
0068 89 f8                  movl %edi,%eax
006a c2 00 00               ret $0
006d
006d                        ovf:
006d 89 ce                  movl %ecx,%esi
006f 8b 76 24               movl 36(%esi),%esi
0072 89 fb                  movl %edi,%ebx
0074 89 1c 24               movl %ebx,(%esp)
0077 bf 00 00 00 00         movl <addOverflow_fast>,%edi
007c 89 f5                  movl %esi,%ebp
007e be 04 00 00 00         movl $4,%esi
0083 ba 19 00 00 00         movl $25,%edx
0088 c7 41 04 04 00 00 00   movl $4,4(%ecx)
008f ff d7                  call *%edi
0091 89 c7                  movl %eax,%edi
0093 eb d0                  jmp call_res
0095
0095                        if_false:
0095 89 cf                  movl %ecx,%edi
0097 8b 7f 24               movl 36(%edi),%edi
009a 83 ec 04               subl $4,%esp
009d 89 3c 24               movl %edi,(%esp)
00a0 bf 00 00 00 00         movl <addGeneral_fast>,%edi
00a5 8b 2c 24               movl (%esp),%ebp
00a8 ba 19 00 00 00         movl $25,%edx
00ad be 04 00 00 00         movl $4,%esi
00b2 c7 41 04 04 00 00 00   movl $4,4(%ecx)
00b9 ff d7                  call *%edi
00bb 83 c4 04               addl $4,%esp
00be 89 c7                  movl %eax,%edi
00c0 eb a3                  jmp call_res
```

Figure 5: x86 assembler for the `add1` function.

Finally, the x86 assembler code (32-bit) produced for the `add1` function is shown in Figure 5. For brevity, this snippet is missing the prelude that adjusts the stack frame if the number of arguments is different than expected. The machine code is generated based on the LIR. The basic blocks are ordered so as to try to linearize the most likely code paths. Basic LIR instructions typically require very few machine instructions. For example, `add_ovf` is implemented as a machine add followed by a jump that tests the overflow condition flag. The x86 code for the `add1` function fits within 194 bytes.

4. RUNTIME

4.1 VM and Program Execution Model

The evolution of dynamic languages shows a trend toward late-binding more and more elements of the language. Accordingly, the implementation of those languages also shows an increasing complexity in their run-time behavior. The availability of the compiler at run-time allows it to manipulate runtime structures required by the program being compiled. This makes for an execution model for the VM that blends compile-time and run-time behaviors. This section explains the particular choices made for the execution model of Tachyon. We introduce the following definitions to simplify reasoning about the execution model in the context of meta-circularity: we refer to the environment in which the compiler executes as the *host* environment, and the environment in which the compiled code executes as the *client* environment.

During execution, a Tachyon program executing in the client environment needs access to a number of data structures. Those data structures are accessed through a context structure. It holds references to the JS global object, a string table and heap allocation pointers. The context structure is an implementation artefact, not accessible as a JS object.

During compilation, the compiler accesses the client environment to initialize resources needed by the compiled code. We chose to create strings in the client environment to avoid maintaining a compile-time string table that would duplicate the runtime string table used by the executing program and avoid the run-time cost of internalizing strings. To be able to inline primitive implementations in the generated code, the compiler needs access to the IR of those primitives. These are maintained in the host environment. Also, the compiler needs access to the OS API functionalities not exposed in JS such as allocating executable memory. This is done through proxies.

The result of a successful compilation is an executable machine code block (MCB), which will be referred after initialization by a JS function object in the client environment. Tachyon also maintains the IR of the function in the host environment to allow recompilation, but this feature has not been used yet. The execution model is illustrated in Figure 6.

Tachyon creates runtime strings during compilation, this technique could also be applied more generally for the creation of runtime function objects maintaining compilation information and manipulation of the global object by the compiler.

When executing on an existing JS implementation, the host environment is necessarily different from the client environment. Once bootstrapped, it would be beneficial to have the host and client environment be the same as this allows sharing of resources between the compiler and compiled code, such as the string table. To keep the implementation simple, however, the compiler currently still executes in a different environment. Sections 4.2 and 5 respectively explain the initialization of the client environment and the bootstrap process.

4.2 Initialization Process

We have designed Tachyon to self-initialize. The host environment compiles code to be run in the client environment. The client code is then able to initialize its own objects in its own heap. The host environment never has direct access to the objects in the client heap. This method was chosen because it avoids the need for an interface layer between the host and client object representations. Only the client code needs to know how to manipulate objects inside its heap.

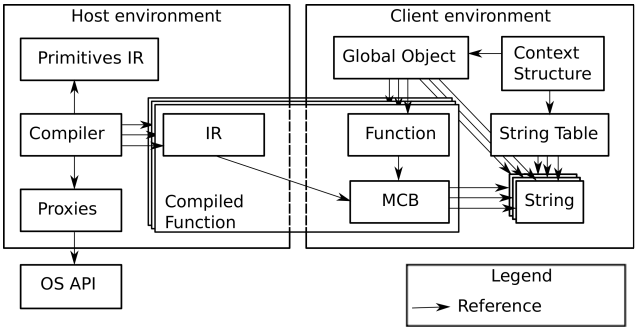

Figure 6: Execution model.

Initialization of the client environment requires a heap in which allocation of objects can occur, a context structure to maintain bookkeeping information and runtime services such as a string table.

The very first operation consists in requesting a contiguous memory space for allocation (heap) from the operating system. This is done through a C `malloc` call.

Next, the compiler needs access to the IR of primitives. To obtain them, the primitives are recompiled from source code and their IR representation are stored on the configuration object in the host environment. This is in turn sufficient to compile bridges between the host environment and the client environment. Bridges reuse the FFI implementation and use C as a common interface language between the host environment and the client environment.

Once the heap, the IR of primitives and bridges are available, the client environment is initialized by allocating the context structure through a FFI call. Since allocation of the context structure requires a context structure, the recursion is avoided by partially initializing the context to allow it to be allocated through the regular allocation mechanism.

Now that object allocation is possible in the client environment, the string table is initialized. Note that all the primitives needed for the previous phases cannot rely on strings for correct behavior since those are not available until this point. However, they might still reference them as long as the code is not executed. At this point, strings used by primitives are allocated in the string table and references are linked in the executable code.

The client global object is then allocated and initialized in the client environment. This allows compilation and initialization of the standard library. Once the standard library is initialized and the system is ready to compile client code, the initialization process is finished.

4.3 Bridges

Tachyon needs to be able to make calls to the client code it compiles in order to initialize it and run it. This process is non-trivial because when Tachyon runs under its host platform, the code it needs to call into uses a calling convention that is not supported by the host platform. Furthermore, even if Tachyon was running independently of a host plat-

form, we may want to change the Tachyon calling convention for a new bootstrap, which would result in a similar scenario. Tachyon may use a different calling convention from the code it is compiling.

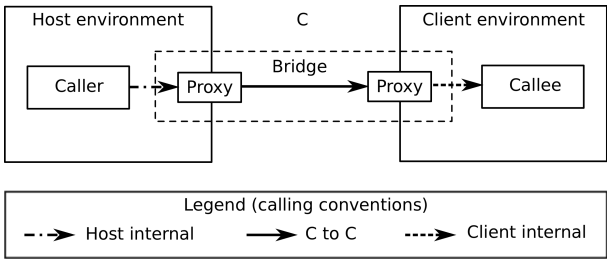

Figure 7: Calling convention bridge.

To resolve this issue, we have designed a mechanism we call a bridge. The Tachyon back-end provides support for calling C functions. It is also able to generate functions that are callable using the C convention. The current Tachyon host platform (Google V8) supports calling into C functions as well. This allows us to implement a system that can call client functions from the host environment by using the C calling convention. We do this by creating a proxy supporting incoming C calls on the client side, and another proxy that exposes the client proxy using the host calling convention on the host side. The host function can then call into its proxy using the host calling convention, which calls the client proxy using the C calling convention, which finally calls into the client function using the client calling convention. This is illustrated in Figure 7.

Bridges are not like the mirrors of Klein [18], which allow reflective access to objects residing in a remote VM. Rather, they are a lower-level mechanism that allows us to call functions residing in another VM while properly handling the discrepancies in calling conventions and argument type conversions. Tachyon uses them to initialize a new VM during the bootstrap process. We may eventually use bridges as a tool in implementing mirror-like facilities.

4.4 Object Representation

Heap-allocated structures in Tachyon are referenced through boxed values whose least significant bits (tag bits) identify the kind of object being referred to. Those structures all begin with a 32-bit header that encodes more precise information about the exact layout and size of the structures so that they can be traversed by a GC.

JS objects are currently represented in memory in a straightforward way. The object structure stores a property count, a prototype reference and an indirect reference to another structure which is a hash map of property names to property values (see Figure 8). This indirect reference is present so that the property map can be reallocated when the number of properties grows beyond the current capacity of the property map. The prototype property refers to the object's prototype object. It may be `null` if the object has no prototype. It is stored outside of the property map because every object must have this field and it must not be directly visible as a property of the object.

In JS, it is possible to use arrays and functions as regular JS objects. That is, named properties can be stored onto

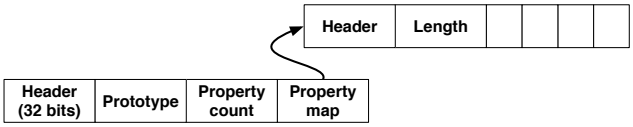

Figure 8: JS object memory layout.

them. They also have a prototype field, just as with regular objects. Because of this, we have chosen to implement arrays and functions as extensions of regular JS objects. This means that arrays and functions share a common part of their memory layout with regular objects. Namely, the prototype, property count and property map fields. They also possess additional fields specific to their implementation.

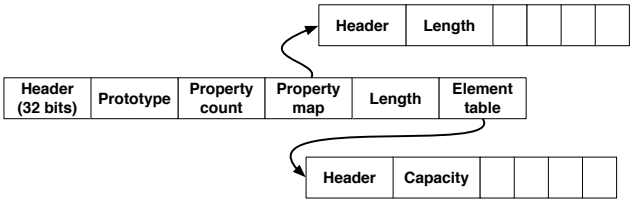

Figure 9: Array memory layout.

In the case of arrays, they also store a length field (the length of the array, or number of indexed values stored) and a reference to an element table (see Figure 9). The element table stores a capacity field so that additional space beyond the length of the array can be reserved for future resizing. This table will be reallocated if the array size increases beyond the capacity.

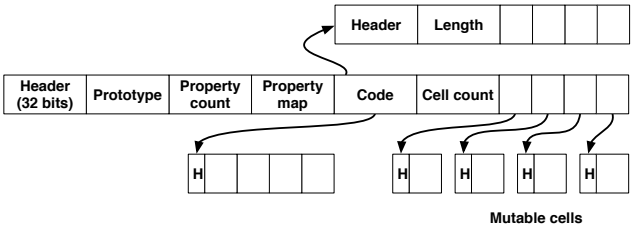

Figure 10: Function memory layout.

Function objects in JS are a representation of closure instances. In addition to the normal object fields, they also store a pointer to the function's machine code and a fixed number of references to mutable cells (see Figure 10). Each mutable cell is heap-allocated and contains a header and a mutable boxed value field. These mutable cells serve to store mutable variables captured by the closure. This representation of closures is one favored by many Scheme implementations. We plan to eventually optimize our closure representation by allocating mutable cells only for the captured variables that are shared among multiple closures.

Strings in JS are not objects. They are immutable primitives, and as such, cannot store properties like objects, arrays and functions. Because of this, we have designed a layout (see Figure 11) for them in which only the length

Figure 11: String memory layout.

and the raw UTF-16 character data are stored.

5. BOOTSTRAP

The boostrap process of Tachyon is performed in memory to avoid the creation of a separate executable or image. This is a temporary measure until support for an image writer is added.

We define the hosted compiler to be the compiler executing in the host environment, the bootstrapped compiler to be the compiler produced by the hosted compiler and executing in the client environment, and the boostrapped client environment to be the environment initialized with the bootstrapped compiler. An illustration of the hosted and bootstrapped compilers is given in Figure 12.

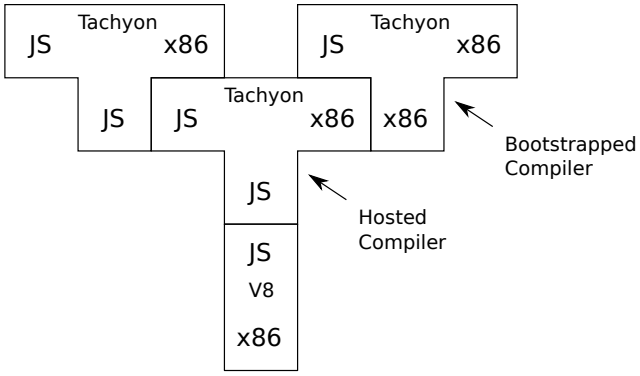

Figure 12: Bootstrap stages.

The following steps are performed to achieve a bootstrap in memory:

1. Initialization of the client environment with the hosted compiler.

2. Compilation of the Tachyon source code and shell with the hosted compiler.

3. Initialization of the bootstrapped client environment with the bootstrapped compiler.

4. Execution of the compiler and shell.

Note that when a bootstrap is performed in memory, the control never returns to the initial host environment, frames from the host environment are kept on the stack and the memory allocated by the host process is still active.

Initialization of the bootstrapped client environment could be avoided if the client environment was reused instead, since their run-time behavior is identical. They were kept separate for simplicity of implementation for this version of the compiler, although we plan on merging them in the near future.

6. PERFORMANCE

In this section, we present some performance comparisons of Tachyon against other JS implementations. Performance has not been an important concern up to this point in Tachyon's development. Therefore, these numbers are meant as ballpark figures only. The benchmark numbers shown were measured on a computer with quad Intel Xeon X5650 CPUs, running the Linux 2.6 kernel. Tachyon[5] was compared against Google V8 revision 7878 and WebKit's interpreter revision 88541. All implementations were compiled in 64-bit mode.

A large proportion of widely-used JS benchmarks available rely on features not yet supported by Tachyon, such as regular expressions, the `Date` object and floating-point numbers. We have chosen to use the benchmarks from the *SunSpider* JS benchmark suite which can run in Tachyon without modification to compare our VM against Google V8 and the WebKit interpreter. The resulting times are shown in Table 1. Because the original benchmarks run very quickly (< 10ms), the times measured are for a total of 400 runs of each benchmark.

Benchmark	Tachyon	Google V8	WebKit Int.
access-binary-trees	20.522	0.473	4.553
access-fannkuck	18.002	2.445	34.871
access-nsieve	4.839	0.686	7.355
bitops-3bit-bits-in-byte	1.890	0.636	9.019
bitops-bits-in-byte	6.301	2.294	11.723
bitops-bitwise-and	30.971	3.436	7.915
bitops-nsieve-bits	7.005	2.347	15.39
controlflow-recursive	6.275	0.739	6.076
crypto-md5	13.789	0.724	7.357
crypto-sha1	15.050	0.870	7.372

Table 1: Running times (in seconds) of *SunSpider* benchmarks under Tachyon, Google V8 and the WebKit interpreter.

The results in Table 1 show that the Tachyon JIT currently produces code that is several times slower on average than that produced by Google V8's JIT. We believe this is largely due to the fact that object property accesses (including global variable accesses and global function calls) are unoptimized in our system. Each such access requires a function call and a hash table lookup on the object. Benchmarks which do not involve property accesses, such as `bitops-3bit-bits-in-byte` are those where Tachyon compares the least unfavorably to V8. Conversely, Tachyon does significantly worse than both V8 and the WebKit interpreter in `access-binary-trees`, a benchmark that is very heavy in terms of property accesses.

At the time of this writing, the Tachyon source code, excluding unit tests and automatically generated parser code, occupies approximately 75 KLOC, compared to around 375 KLOC for V8 and 550 KLOC for SpiderMonkey. Tachyon is clearly still in its infancy, but since it is a large and complex piece of software, we have decided to also use the time it takes Tachyon to compile itself as a benchmark. We believe this is a more representative measure of Tachyon's performance than the JS microbenchmarks widely used today.

Compilation times for Tachyon, first running under V8 and then running under itself are given in Table 2. These

[5]https://github.com/Tachyon-Team/Tachyon/tree/dls2011

Benchmark	Tachyon	Google V8
Tachyon compilation time	1991	165

Table 2: Compilation times (in seconds) for Tachyon under Tachyon and Google V8.

numbers indicate that Tachyon's overall performance is about an order of magnitude slower when compiled by itself than when running under V8.

We believe that these numbers are encouraging. Tachyon, despite its limitations, does significantly better than WebKit's bytecode interpreter on several benchmarks. We have already started work to improve Tachyon's performance. For instance, we have started prototyping a code patching mechanism to optimize accesses to globals. The substantial performance improvements obtained encourage us to explore other "low-hanging fruit" optimizations. We believe that we may soon reach a level of performance that is more competitive with that of V8.

7. FUTURE WORK

The Tachyon bootstrap currently occurs inside a custom heap allocated inside the Google V8 process. As such, if one wants to execute Tachyon while bootstrapped, we currently have to begin the bootstrap compilation of Tachyon anew. We are currently working on the implementation of a memory dump of the Tachyon heap (all machine code and heap data) into an ELF binary image file. This will allow Tachyon to become truly independent of its host platform.

Tachyon currently has no garbage collector. It has been built with a compacting, generational GC in mind, but this GC is not yet complete. This currently imposes a limit on the amount of memory programs can allocate during their execution. Work has begun on the implementation of a GC. This collector will initially be a single-threaded compacting GC, and will be written in plain C, because of the wide availability of debugging tools. However, we plan to eventually rewrite this in our extended JS dialect. We believe this will make the GC easier to maintain in the long run, as C code does not have direct access to the definitions of memory layouts used by our heap-allocated objects.

Our current object representation is very simple. Each object stores a hash map of property names to property values. While easy to implement, this is inefficient both in terms of running-time and space usage. As such, we plan to factor out the name of properties and their position by introducing a layout object, called maps in SELF [3] and hidden classes in V8. An object then keeps a reference to its layout object and only stores the values of its properties. Layout objects can be shared by multiple objects with the same layout.

The subset of ES5 supported by Tachyon has proven sufficient to compile Tachyon itself. A few features are still missing, however. Namely floating-point numbers, exceptions, regular expressions, `eval` and object property attributes. We do not foresee any difficulties in implementing the missing features. The compiler has been designed with support for features like `eval` in mind. Once these missing features are implemented, we believe Tachyon will be able to run most JS benchmarks currently available.

Besides getting Tachyon to support all of ES5, one of our medium-term goals is to bring its performance to a competitive level. Our project has thus far been mostly focused on rapidly achieving a working bootstrap compilation. However, we believe it is important for our compiler to generate quality code if we are to compare it to other existing implementations. As such, we aim to reach a performance level within a factor of two of the best existing JS implementations within the next year. Another performance goal is to improve the speed at which Tachyon compiles source code. This will be partly achieved by having Tachyon generate better code when compiling itself.

Since Tachyon is being developed as a research platform, we intend to use it to experiment with novel ideas. One area we plan to explore is the use of more aggressive speculative optimizations. We intend to test the concept of "optimistic" optimizations: optimizations that are likely to be safe given the current state of a running program, but are not guaranteed safe for its entire execution. Such opportunities for optimizations can be discovered using combination of profiling and static analysis. Aggressively optimized code will then be generated under the optimistic assumption that the said optimizations will remain applicable, but guards need to be inserted in the code so that it can be deoptimized should this assumption be invalidated.

Tachyon currently runs inside of the Google V8 shell, which is a console program. This allows us to implement the ES5 specification, but does not allow us to use Tachyon inside a web browser. Since the main use for JS at this time is within web pages, it is one of our main goals to eventually integrate Tachyon into a web browser of some sort. Mozilla Corp. has expressed interest in implementing an HTML DOM tree in JS for an experimental web browser. We believe this may be a very interesting platform for Tachyon to integrate into. We are also looking at *node.js* as a possible alternative.

8. CONCLUSION

JS is currently one of the most widespread dynamic languages. As web applications become more complex, JS performance is becoming increasingly important. Existing JS engines, even when they are open-source, are difficult to modify. A flexible research platform is therefore needed in order to design, implement and evaluate new compilation techniques for JS. We have presented Tachyon, a self-hosted JS virtual machine that aims to fill this void. Tachyon is itself written in an extended dialect of JS. We have shown that this implementation decision allows the system to be quickly developped and easily modified. Tachyon is thus well-suited to rapid prototyping of new compilation strategies. For instance, it already supports two different register allocators and two target architectures. Tachyon currently supports a subset of the full ES5 specification that is sufficient to enable the bootstrapping process.

We have shown that the current version of Tachyon, despite not having been optimized for performance of the generated code, is faster than the WebKit interpreter on some benchmarks from the popular *SunSpider* suite. We believe that, once some straightforward optimizations are added, the performance of Tachyon will be competitive with existing JS JIT-based VMs. The flexibility of the compiler will also allow deeper optimizations to be investigated.

Tachyon is publicly available under a Modified BSD license on GitHub. Contributions are welcome!

9. ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and Mozilla Corporation.

We wish to thank David Haguenauer, Éric Thivierge, Olivier Matz and Alexandre St-Aubin for reviewing drafts of this paper.

10. REFERENCES

[1] C. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 2009 workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 2004 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.

[3] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM.

[4] D. Crockford. *JavaScript: The Good Parts*, chapter 4, page 42. O'Reilly, 2008.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 1989 ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478. ACM, 2009.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32, pages 318–326. ACM, ACM Press, 1997.

[8] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 IEEE/ACM international symposium on Code generation and optimization*, 0:75, 2004.

[10] F. Logozzo and H. Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation–Application to JavaScript Optimization. In *Proceedings of the 2010 international conference on Compiler construction*, pages 66–83. Springer, 2010.

[11] F. Loitsch. JavaScript to Scheme compilation. In *Proceedings of the 2005 Workshop on Scheme and Functional Programming*, pages 101–116, september 2005.

[12] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, September 1999.

[13] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, page 3. USENIX Association, 2010.

[14] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming*. Springer, 2011.

[15] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2010.

[16] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 2006 ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[17] R. Sol, C. Guillon, F. M. Q. a. Pereira, and M. A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 2011 international conference on Compiler construction*, pages 2–21, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[19] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[20] C. Wimmer and M. Franz. Linear scan register allocation on SSA form. In *Proceedings of the 2010 IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, New York, NY, USA, 2010. ACM.

CHAPTER 4

EAGER BASIC BLOCK VERSIONING

In this chapter, we present the first article on BBV we have submitted for publication. The final, published version of the article is found in Chapter 5. We have included this original version of our first BBV publication because the technique we developed, and its performance characteristics, went on to evolve significantly during the year between the first and final submissions. The article, *Removing Dynamic Type Tests with Context-Driven Basic Block Versioning*, explains the core ideas behind what we now refer to as *eager* BBV. The article presented in this chapter is publicly available through arxiv.org [5].

4.1 Problem and Motivation

As discussed in previous chapters, an implementation of a dynamically typed programming language invariably contains implicit dynamic type checks as part of its semantics. Most of these type tests are redundant, simply because most variables and object fields within a program do not change type at run time. As such, we would like to eliminate as many redundant type tests as possible in order to maximize performance.

The traditional way to eliminate dynamic type checks is to do program analysis, either at the intraprocedural, or at the interprocedural (whole-program) level. This is problematic for multiple reasons:

— Program analyses are expensive, both in terms of execution time and memory.

— JIT compilers must pay for the overhead of optimization during program execution, and may run on resource-constrained devices such as cellphones.

— Program analyses are limited in precision. There is a fundamental limit to the amount of information that can be inferred about a program without executing it, and trying to extract more precise information inevitably comes at significantly increased analysis cost.

The 2010 article *Interprocedural Analysis with Lazy Propagation* [26] illustrates the potentially high overhead of whole-program static type analyses, with program analysis times up to over two minutes, and memory usage potentially up to 140 megabytes, for benchmarks programs which are only a few hundred lines long. This analysis uses context-sensitivity to try and increase precision, but still suffers important limitations due to its ahead of time nature.

Program analyses running in ahead of time compilers struggle to predict the behavior of running programs, in part because they lack data, as they do not have access to program inputs. In contrast, JIT compilers offer an advantage which should be immensely powerful: they can observe programs and the data they manipulate at execution time. A JIT compiler not only has access to a program's input, it should, in principle, be able to observe the type of any variable at any point during a program's execution.

Previous work on tracing JIT compilers is based on this principle. A tracing JIT only compiles the parts of a program that are executed. Tracing JITs are also able to perform simple analyses on long traces (recorded linear sequence of instructions). Among other things, they can perform some simple type propagation. This is inexpensive, and it is based on sequences of instructions that are actually executed at run time, as opposed to an ahead of time prediction of what might be executed.

The literature on tracing compilation, particularly the papers written about Mozilla's TraceMonkey compiler, seemed very enthusiastic about the technique, painting it as a JIT compilation miracle. Unfortunately, our own efforts at implementing a tracing JIT compiler, and meetings with Mozilla compiler engineers, painted a different picture. TraceMonkey was retired in 2011, in part because of limitations of the tracing algorithm it implemented. One of the main flaws of this technique was that it dealt very poorly with loops containing many unpredictable branches, which caused it to compile an exponentially large number of traces.

My previous M.Sc. work on McVM was based on the idea of a JIT compiler intercepting information about argument types when function calls occur, and then using this information to compile one or more type-specialized versions of a function [9]. This type information would be fed into an intraprocedural fixed-point type analysis to eliminate

type checks on local variables.

McVM was able to perform some degree of interprocedural type specialization at relatively low cost, but its ability to eliminate local type checks was still constrained by the precision limitations of intraprocedural type analyses. McVM was also unable to propagate type information through recursive function calls.

4.2 Basic Block Versioning

BBV is a technique that operates at the level of individual basic blocks. A basic block is a single-entry single-exit sequence of IR instructions, terminated by a branch instruction that may jump to one or more successor basic blocks. Branch instructions include unconditional jumps, conditional jumps as well as type test primitives. The `ret` (return) and `throw` instructions are also considered branch instructions.

The technique, as presented here, clones basic blocks speculatively at method compilation time. The basic blocks are cloned so that they can be type-specialized. That is, each basic block is specialized based on the types of live variables at the entry of the said block. Basic block versioning is sensitive to control-flow, and type test primitives are used to accumulate type information and propagate this information to successor blocks.

BBV can be seen as a form of controlled, systematic tail duplication, or an unfolding of the control flow graph. It is similar to a type analysis, but instead of computing a fixed point on the types of variables, it is computing a fixed point on the generation of new block versions. Individual block versions are keyed based on the types of variables live at the block entry. New block versions cease to be created when no new variable type combinations occur for a given block, or when a hard limit on the number of versions for any given block is reached.

Figures 4.1 and 4.2 illustrate the tail splitting and type propagation performed by BBV at a small scale. In Figure 4.1, a test is performed to determine if variable `n` has type `int32` or not. We can infer that if we reach block `A`, then `n` must be `int32`. Correspondingly, if we reach block `B`, then `n` cannot be `int32`. Unfortunately, in block

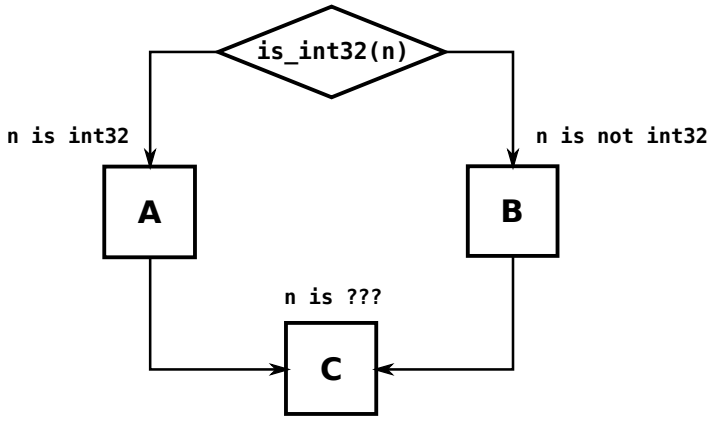

Figure 4.1: Type test and control-flow merge without BBV.

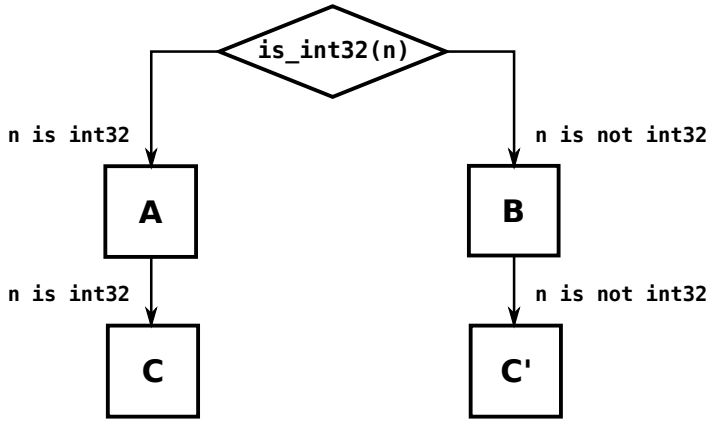

Figure 4.2: Type test and tail splitting with BBV.

`C`, we have a control flow merge, and we know nothing about the type of `n`. In Figure 4.2, we have created two versions of block `C`, such that type information about `n` can be preserved in each of them.

4.3 Results

The article demonstrates that the number of type tests performed on benchmarks can be significantly reduced. This is done at the cost of a code size increase, but the increase is shown to be moderate. The main flaw of the approach, however, is that

despite important reductions in the number of type tests, we were unable, with this early version of the BBV algorithm, to show execution time improvements.

Eager basic block versioning succeeds in reducing the number of type checks, but increases code size too much in some cases, resulting in a higher occurrence of cache misses. A key insight, the introduction of laziness, improves the performance of BBV on all metrics, and enables execution time as well as precision improvements. This is discussed in Chapter 5.

Removing Dynamic Type Tests with Context-Driven Basic Block Versioning

Maxime Chevalier-Boisvert and Marc Feeley

DIRO, Université de Montréal, Montreal, QC, Canada

Abstract. Dynamic typing is an important feature of dynamic programming languages. Primitive operators such as those for performing arithmetic and comparisons typically operate on a wide variety of input value types, and as such, must internally implement some form of dynamic type dispatch and type checking. Removing such type tests is important for an efficient implementation.
In this paper, we examine the effectiveness of a novel approach to reducing the number of dynamically executed type tests called *context-driven basic block versioning*. This simple technique clones and specializes basic blocks in such a way as to allow the compiler to accumulate type information while machine code is generated, without a separate type analysis pass. The accumulated information allows the removal of some redundant type tests, particularly in performance-critical paths.
We have implemented intraprocedural context-driven basic block versioning in a JavaScript JIT compiler. For comparison, we have also implemented a classical flow-based type analysis operating on the same concrete types. Our results show that basic block versioning performs better on most benchmarks and removes a large fraction of type tests at the expense of a moderate code size increase. We believe that this technique offers a good tradeoff between implementation complexity and performance, and is suitable for integration in production JIT compilers.

1 Introduction

Dynamic programming languages make heavy use of late binding in their semantics. In essence this means doing at run time what can be done before run time in other programming languages, for example type checking, type dispatch, function redefinition, code linking, program evaluation (e.g. `eval`), and compilation (e.g. JIT compilation). In dynamic programming languages such as JavaScript, Python, Ruby and Scheme, there are no type annotations on variables and types are instead associated with values. Primitive operations, such as `+`, must verify that the operand values are of an acceptable type (type checking) and must use the types of the values to select the operation, such as integer addition, floating point addition, or string concatenation (type dispatching). We will use the generic term *type test* to mean a run time operation that determines if a value belongs to a given type. Type checking and dispatching are built with type tests.

VMs for dynamic programming languages must tackle the run time overhead caused by the dynamic features to achieve an efficient execution. Clever type

representation and runtime system organization can help reduce the cost of the dynamic features. In this paper we focus on reducing the number of type tests executed, which is a complementary approach.

Static type analyses which infer a type for each variable can help remove and in some cases eliminate type test cost. However, such analyses are of limited applicability in dynamic languages because of the run time cost and the presence of generic operators. A whole program analysis provides good precision, compared to a more local analysis, but it is time consuming, which is an issue when compilation is done during program execution. Moreover, the results are generally invalidated when functions are redefined and code is dynamically loaded or evaluated, requiring a new program analysis. This often means that analysis precision must be traded for speed. Intraprocedural analyses are a good compromise when such dynamic features are used often, the program is large, the running time is short or a simple VM design is desired. The complexity of the type hierarchy for the numerical types may negatively impact the precision of the type analysis due to its conservative nature. To implement the numerical types of the language or for performance, a VM may use several concrete types for numbers (e.g. fixed precision integers, infinite precision integers, floating point numbers, complex numbers, etc). The VM automatically converts from one representation to another when an operator receives mixed-representation operands or limit cases are encountered (e.g. overflows). Variables containing numbers will often have to be assigned a type which is the union of some concrete numerical types (e.g. `int` $\cup$ `float`) if they store the result of an arithmetic operator. This means that a type dispatch will have to be executed when this variable is used as an operand of another arithmetic operator. This is an important issue due to the frequent use of arithmetic in typical programs (for example an innocuous looking `i++` in a loop will typically require a type dispatch and overflow check).

We propose a new approach which reduces the number of type tests by eliminating those that are redundant within each function. Basic block versioning aims to be a simple and efficient technique mixing code generation, analysis and optimization. Section 2 explains the basic block versioning approach in more details. An implementation of our approach in a JavaScript compiler is described in Section 3 and evaluated in Section 4. Related work is presented in Section 5.

2 Basic Block Versioning

```
function f(n) {
    if (n<0) n = n+1;
    return n+1;
}
```

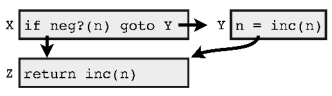

Fig. 1. Definition for function `f` and the corresponding high-level control flow graph.

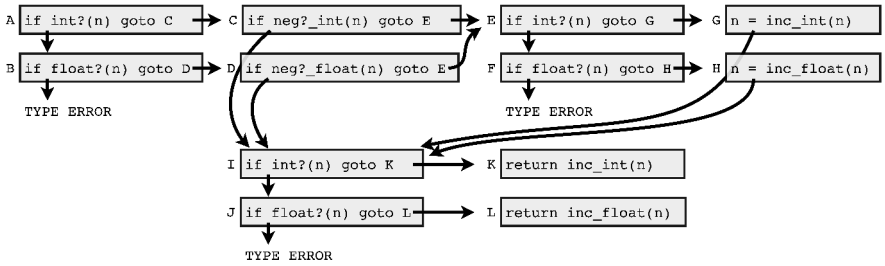

Fig. 2. Control flow graph after the inlining of the primitive operators `neg?` and `inc`.

The basic block versioning approach generates one or more versions of each live basic block based on type information derived from the type tests executed by the code. The type analysis and code generation are performed together, generating on-demand new versions of blocks specialized to the typing context of predecessor blocks.

An important difference between this approach and traditional type analyses is that basic block versioning does not compute a fixed-point on types, but rather computes a fixed-point on the generation of new block versions, each associated with a configuration of incoming types. Values which have different types at the same program point are handled more precisely with basic block versioning. In a traditional type analysis the union of the possible types would be assigned to the value, causing the analysis to be conservative. With basic block versioning, distinct basic blocks will be created for each type tested previously, allowing a more precise tracking of types. Because versions are created on demand, only versions for the relevant type combinations are created.

To illustrate this approach we will use a simple example in a hypothetical dynamically typed language similar only in syntax to JavaScript. Consider the function `f` whose definition and corresponding high-level control flow graph are shown in Figure 1. Lets assume that there are only two concrete types for numbers: `int`, a fixed precision integer, and `float`, a floating point number.[1] The value of parameter `n` must be one of these two types, otherwise it is a type error. The primitive operations `neg?(n)` and `inc(n)` must include a type dispatch to select the appropriate operation based on the concrete type of `n`. Inlining these primitive operations makes the type tests explicit as shown in the control flow graph in Figure 2. Note that basic block X has been expanded to basic blocks A-D, while Y has been expanded to E-H, and Z has been expanded to I-L. Note that for simplicity we will assume that the `inc_int(n)` operation yields an `int` (i.e. there is no overflow check).

Basic block versioning starts compiling basic block A with a context where value `n` is of an unknown type. This will generate the code for the `int?(n)`

[1] Note that JavaScript has a single type for numbers, which corresponds to IEEE 64-bit floating point numbers, but an implementation of JavaScript could implement numbers with these two concrete types to benefit from the performance of integer arithmetic for integer loop iteration variables and array indexing.

type test and will schedule the compilation of a version of block B, called $B_{\neg \texttt{int}}$, where the value `n` is known to not be an `int` and will schedule the compilation of a version of block C, called $C_{\texttt{int}}$, where the value `n` is known to be an `int`. Our use of subscripts is a purely notational way of keeping track of the type context information, which only needs to give information on `n` in this example. When basic block $C_{\texttt{int}}$ is compiled, code is generated for the `neg?_int(n)` test and this schedules the compilation of versions of blocks E and I, called $E_{\texttt{int}}$ and $I_{\texttt{int}}$ respectively, where the value `n` is known to be an `int`. Note that the compilation of $B_{\neg \texttt{int}}$ will cause the compilation of $D_{\texttt{float}}$, which will also schedule the compilation of versions of blocks E and I but in a different context, where the value `n` is known to be a `float` (blocks $E_{\texttt{float}}$ and $I_{\texttt{float}}$ respectively).

The type tests in the four blocks $E_{\texttt{int}}$, $E_{\texttt{float}}$, $I_{\texttt{int}}$ and $I_{\texttt{float}}$ can be removed and replaced by direct jumps to the appropriate destination blocks. For example $E_{\texttt{int}}$ becomes a direct jump to $G_{\texttt{int}}$ and $E_{\texttt{float}}$ becomes a direct jump to $F_{\texttt{float}}$. Because $G_{\texttt{int}}$ and $H_{\texttt{float}}$ jump respectively to $I_{\texttt{int}}$ and $I_{\texttt{float}}$, the type tests in those blocks are also removed. Note that the final generated code implements the same control flow graph as Figure 3. Two of the three type dispatch operations in the original code have been removed.

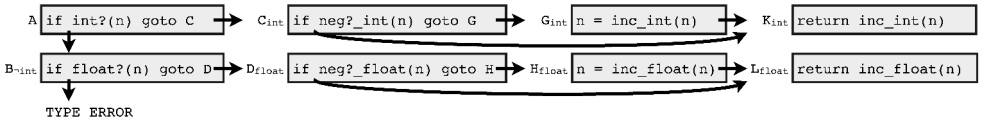

Fig. 3. Final control flow of function `f` after basic block versioning.

Let us now consider what would happen if the `inc_int(n)` operations detected integer overflow and yielded a `float` result in that case. Then the compilation of basic block $G_{\texttt{int}}$ would schedule two versions of the successor basic block I: $I_{\texttt{int}}$ for the case where there is no overflow, and $I_{\texttt{float}}$ for the case where there is an overflow. Due to the normal removal of type tests, when `inc_int(n)` would overflow, a `float` would be stored in `n` followed by a direct jump to block $L_{\texttt{float}}$. Thus the only change in the control flow of Figure 3 is that block $G_{\texttt{int}}$ has an edge to $L_{\texttt{float}}$ in addition to $K_{\texttt{int}}$. So here too a single type dispatch is needed in function `f`.

In theory, the number of possible type configurations in a context grows combinatorially with the number of live values whose type is being accounted for and the number of types they can have. We believe that a combinatorial explosion is unlikely to be a problem in practice because typically the number of values live at a given program point is small and the number of possible types of a value is small.

There are pathological cases where a large number of block versions are needed to account for all the possible incoming type combinations. To prevent such occurrences, a simple approach is to place an arbitrary limit per block on the number of versions that are compiled. Once this limit is hit for a given block,

a general version of the block will be compiled, which makes no assumptions about incoming types, i.e. all values are of unknown type.

If more versions of a block would be required than the block version limit allows, it is advantageous to compile the versions which will be executed most often. This can be done by monitoring the frequency of execution of each basic block (with a counter per block) prior to the JIT compilation. Generating linear machine code sequences along hot paths first has the beneficial effect that it will tend to prioritize the compilation of block versions for type combinations that occur more frequently at run time. This strategy is used in our experiments.

An indirect benefit of our basic block versioning approach is that it automatically unrolls some of the first iterations of loops in such a way that type tests are hoisted out of loop bodies. For example, if variables of unknown type are used unconditionnaly in a loop, their type will be tested inside the first iteration of the loop. Once this test is performed in the first loop iteration, the type information gained will allow the loop body to avoid the redundant type tests for the remaining iterations.

3 Implementation in Higgs

We have implemented basic block versioning inside a JavaScript virtual machine called Higgs. This virtual machine comprises an interpreter and a JIT compiler targeted at x86-64 POSIX platforms. The current implementation of Higgs supports most of the ECMAScript 5 specification [1], with the exception of the `with` statement, property attributes and getter-setter properties. Its runtime and standard libraries are self-hosted, written in an extended dialect of ECMAScript with low-level primitives. These low-level primitives are special IR instructions which allow us to express type tests as well as integer and floating point machine instructions in the implementation language.

In Higgs, the interpreter is used for profiling, and as a default, unoptimized mode of execution. Functions are parsed into an abstract syntax tree, and lazily compiled to an Static Single Assignment (SSA) Intermediate Representation (IR) when they are first called. The interpreter then executes code in SSA form directly. As code is executed by the interpreter, counters on basic blocks are incremented every time a given block is executed. Frequency counts for each potential callee are also collected at call sites.

The JIT compiler is triggered when the execution count for a function entry block or loop header reaches a fixed threshold (currently set to 800). Callees are first aggressively inlined into the function to be compiled. This is done by substituting the IR of callees at call sites. Calls are currently inlined only if profiling data indicates that they are monomorphic, and the callee is 30 basic blocks or less, which enables inlining of most runtime primitives. Call sites belonging to blocks with higher execution frequencies are prioritized for inlining. Once inlining is complete, the fused IR containing inlined callees is then optimized using simple subgraph substitution patterns before machine code generation proceeds.

Algorithm 1 Code generation with basic block versioning

```
1: procedure genFun(assembler, function)
2:     workList ← ∅                                   ▷ Stack of block versions to be compiled
3:     versionList ← ∅                                ▷ List of existing block versions
4:     getLabel(function.entryBlock, ∅, workList, versionList)    ▷ Begin compilation
5:     while workList not empty do
6:         block, ctx, label ← workList.pop()
7:         assembler.addLabel(label)                  ▷ Insert the label for this block
8:         if block.execCount is 0 then
9:             genStub(assembler, block, ctx, label)
10:        else
11:            for instr in block.instrs do           ▷ Generate code for each instruction
12:                genInstr(assembler, instr, ctx, workList, versionList);
13:            end for
14:        end if
15:    end while
16: end procedure
17: procedure getLabel(block, ctx, workList, versionList)
18:    if numVersions(block) ≥ maxvers then    ▷ If the version limit for this block
    was reached
19:        bestMatch ← findBestMatch(block, ctx, versionList);
20:        if bestMatch ≠ null then                   ▷ If a compatible match was found
21:            return bestMatch
22:        else
23:            ctx ← ∅        ▷ Make a generic version accepting all incoming contexts
24:        end if
25:    end if
26:    label ← newLabel();
27:    workList.push(⟨block, ctx, label⟩);      ▷ Queue the new version to be compiled
28:    versionList.append(⟨block, ctx, label⟩); ▷ Add the new block version to the list
29:    return label
30: end procedure
```

Algorithm 2 Code generation with basic block versioning

```
31: procedure AddInt32.genInstr(assembler, instr, ctx, workList, versionList)
32:     assembler.addInt32(instr.getArgs())    ▷ Generate the add machine instruction
33:     ctx.setOutType(instr, int32) ▷ The output type of AddInt32 is always int32. If
    an overflow occurs, the result is recomputed using AddFloat64
34: end procedure
35: procedure IsInt32.genInstr(assembler, instr, ctx, workList, versionList)
36:     argType ← ctx.getType(instr.getArg(0))
37:     if argType is int32 then
38:         ctx.setOutType(instr, true)
39:     else if argType ≠ ⊤ then
40:         ctx.setOutType(instr, false)
41:     else
42:         assembler.isInt32(instr.getArgs())          ▷ Generate code for the type test
43:         ctx.setOutType(instr, const)
44:     end if
45: end procedure
46: procedure Jump.genInstr(assembler, instr, ctx, workList, versionList)
47:     label ← getLabel(instr.target, ctx, workList, versionList)
48:     assembler.jump(label)
49: end procedure
50: procedure IfTrue.genInstr(assembler, instr, ctx, workList, versionList)
51:     arg ← instr.getArg(0)
52:     argType ← ctx.getType(arg)
53:     trueCtx ← ctx.copy()                          ▷ New context for the true branch
54:     if arg instanceof IsInt32 then
55:         trueCtx.setType(arg.getArg(0), int32)
56:     end if
57:     if argType is true then
58:         trueLabel ← getLabel(instr.trueTarget, trueCtx, workList, versionList)
59:         assembler.jump(trueLabel)
60:     else if argType is false then
61:         falseLabel ← getLabel(instr.falseTarget, ctx, workList, versionList)
62:         assembler.jump(falseLabel)
63:     else
64:         trueLabel ← getLabel(instr.trueTarget, trueCtx, workList, versionList)
65:         falseLabel ← getLabel(instr.falseTarget, ctx, workList, versionList)
66:         assembler.compare(arg, true)              ▷ Compare the argument to true
67:         assembler.jumpIfEqual(trueLabel)
68:         assembler.jump(falseLabel)
69:     end if
70: end procedure
```

Machine code generation (see Algorithm 1) begins with the function's entry block and entry context pair being pushed on top of a stack which serves as a work list. This stack is used to keep track of block versions to be compiled, and enable depth-first generation of hot code paths. Code generation proceeds by repeatedly popping a block and context pair to be compiled off the stack. If the block to be compiled has an execution count of 0, stub code is generated out of line, which spills live variables, invalidates the generated machine code for the function and exits to the interpreter. Otherwise, code is generated by calling code generation methods corresponding to each IR instruction to be compiled in the current block, in order.

As each IR instruction in a block is compiled, information is both retrieved from and inserted into the current context. Information retrieved may be used to optimize the compilation of the current instruction (e.g. eliminate type tests). Instructions will also write their own output type in the context if known. The last instruction of a block, which must be a branch instruction, may potentially push additional compilation requests on the work stack. More specifically, branch instructions can request an assembler label for a version of a block corresponding to the current context at the branch instruction. If such a version was already compiled, the label is returned immediately. Otherwise, a new label is generated, the block and the current context are pushed on the stack, to be compiled later.

To avoid pathological cases where a large number of versions could be generated for a given basic block, we limit the number of versions that may be compiled. This is done with the `maxvers` parameter, which specifies how many versions can be compiled for any single block. Once this limit is hit for a particular block, requests for new versions of this block will first try to find if an inexact but compatible match for the incoming context can be found. An existing version is compatible with the incoming context if the value types assumed by the existing version are the same as, or supertypes of, those specified in the incoming context. If a compatible match is found, this match will be returned. If not, a generic version of the block will be generated, which can accept all incoming type combinations. When the `maxvers` parameter is set to zero, basic block versioning is disabled, and only the generic version is generated.

3.1 Type tags and runtime primitives

The current version of Higgs segregates values into a few categories based on type tags [13]. These categories are: 32-bit integers (`int32`), 64-bit floating point values (`float64`), garbage-collected references inside the Higgs heap (`refptr`), raw pointers to C objects (`rawptr`) and miscellaneous JavaScript constants (`const`). These type tags form a simple, first-degree notion of types which we use to drive the basic block versioning approach. The current implementation of basic block versioning in Higgs does not differentiate between references to object, arrays and functions, but instead lumps all of these under the reference pointer category. We do, however, distinguish between the boolean `true` and `false` constants to enable the propagation of type test results.

We believe that this choice of a simple type representation is a worthwhile way to investigate the effectiveness and potential of basic block versioning. Higgs implements JavaScript operators as runtime library functions written in an extended dialect of JavaScript, and most of these functions use type tags to do dynamic dispatch. As such, eliminating this first level of type tests is crucial to improving the performance of the system as a whole. Extending the system to use a more precise representation of types is part of future work.

```
function $rt_add(x, y) {
    if ($ir_is_i32(x)) { // If x is integer
        if ($ir_is_i32(y)) {
            if (var r = $ir_add_i32_ovf(x, y))
                return r;
            else // Handle the overflow case
                return $ir_add_f64($ir_i32_to_f64(x),
                                   $ir_i32_to_f64(y));
        } else if ($ir_is_f64(y))
            return $ir_add_f64($ir_i32_to_f64(x), y);
    } else if ($ir_is_f64(x)) { // If x is floating point
        if ($ir_is_i32(y))
            return $ir_add_f64(x, $ir_i32_to_f64(y));
        else if ($ir_is_f64(y))
            return $ir_add_f64(x, y);
    }

    // Evaluate arguments as strings and concatenate them
    return $rt_strcat($rt_toString(x), $rt_toString(y));
}
```

Fig. 4. Implementation of the + operator

Figure 4 illustrates the implementation of the primitive + operator. As can be seen, this function makes extensive use of low-level type test primitives such as `$ir_is_i32` and `$ir_is_f64` to implement dynamic dispatch based on the type tags of the input arguments. All other arithmetic and comparison primitives implement a similar dispatch mechanism.

3.2 Flow-based representation analysis

To provide a point of comparison and contrast the capabilities of basic block versioning with that of more traditional type analysis approaches, we have implemented a forward flow-based representation analysis which computes a fixed-point on the types of SSA values. The analysis is an adaptation of Wegbreit's algorithm as described in [26]. It is an intraprocedural constant propagation analysis which propagates the types of SSA values in a flow-sensitive manner.

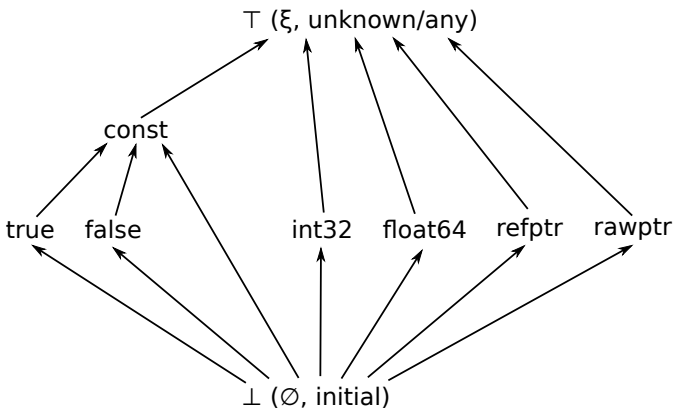

Fig. 5. Type lattice used by the representation analysis

Pseudocode for this analysis and some of its transfer functions is shown in Appendix A.

The representation analysis uses the same type representation (see Figure 5) as our basic block versioning implementation, and has similar type analysis capabilities. It is able to gain information from type tests and forward this information along branches. It is also able to deduce, in some cases, that specific branches will not be executed and ignore the effects of code that was determined dead.

We have also extended the flow-based algorithm to ignore basic blocks which are unexecuted (have an execution count of 0) at analysis time. This allows the analysis to ignore some code paths not executed up to now, which is useful in some cases, since primitive language operators often have multiple paths which can result in different output types. If presumed dead blocks turn out to be executed later, analysis results and associated compiled code will be invalidated at run time. This was done to make the analysis more competitive with basic block versioning which by construction ignores stubbed blocks, for which no compiled code was generated.

3.3 Limitations

There are a few important limitations to the current implementation of basic block versioning in Higgs. We do not, at this point, track the types of object properties. Global variables, which are properties of the global object in JavaScript, are also untracked. We do not account for interprocedural flow of type information either. That is, function parameter and return value types are assumed to be unknown. Finally, the current implementation of Higgs does not implement any kind of load-store forwarding optimization. These limitations are nontrivial to tackle due to factors such as the late-bound nature of JavaScript, the potential presence of the `eval` construct, dynamic addition and deletion of properties and the dynamic installation of getter-setter methods on object fields.

The results presented in this paper are entirely based on an intraprocedural implementation of basic block versioning which accounts only for the types of local variables and temporaries, in combination with aggressive inlining of library and method calls. Extending basic block versioning to take object identity, array and property type information into account constitutes future work.

4 Evaluation

To assess the effectiveness of basic block versioning, we have used a total of 24 benchmarks from the classic SunSpider and Google V8 suites. A handful of benchmarks from both suites were not included in our tests because the current Higgs implementation does not yet support them.

Figure 6 shows counts of dynamically executed type tests across all benchmarks for the representation analysis and for basic block versioning with various block version limits. These counts are relative to a baseline which has the version limit set to 0, and thus only generates a default, unoptimized version of each basic block, without attempting to eliminate any type tests. As can be seen from the counts, the analysis produces a reduction in the number of dynamically executed type tests over the unoptimized default on every benchmark. The basic block versioning approach does at least as well as the analysis, and almost always significantly better. Surprisingly, even with a version cap as low as 1 version per basic block, the versioning approach is often competitive with the representation analysis.

Raising the version cap reduces the number of tests performed with the versioning approach in a seemingly asymptotic manner as we get closer to the limit of what is achievable with our implementation. The versioning approach does remarkably well on the `bits-in-byte` benchmark, with a reduction in the number of type tests by a factor of over 50. This benchmark (see Figure 7) is an ideal use case for our versioning approach. It is a tight loop performing bitwise and arithmetic operations on integers which are all stored in local variables. The versioning approach performs noticeably better than the analysis on this test because it is able to hoist a type test on the function parameter `b` out of a critical loop. The type of this parameter is initially unknown when entering the function. The analysis on its own cannot achieve this, and so must repeat the test every loop iteration. Note that neither the analysis nor the basic block versioning approach need to test the type of `c` at run time because the variable is initialized to an integer value before loop entry, and integer overflow never occurs, so the overflow case remains a stub. The `bitwise-and` benchmark operates exclusively on global variables, for which our system cannot extract types, and so neither the type analysis nor the versioning approach show any improvement over baseline for this benchmark.

A breakdown of relative type test counts by kind, averaged accross all benchmarks (using the geometric mean) is shown in Figure 8. We see that the versioning approach is able to achieve better results than the representation analysis across each kind of type test. The `is_refptr` category shows the least improve-

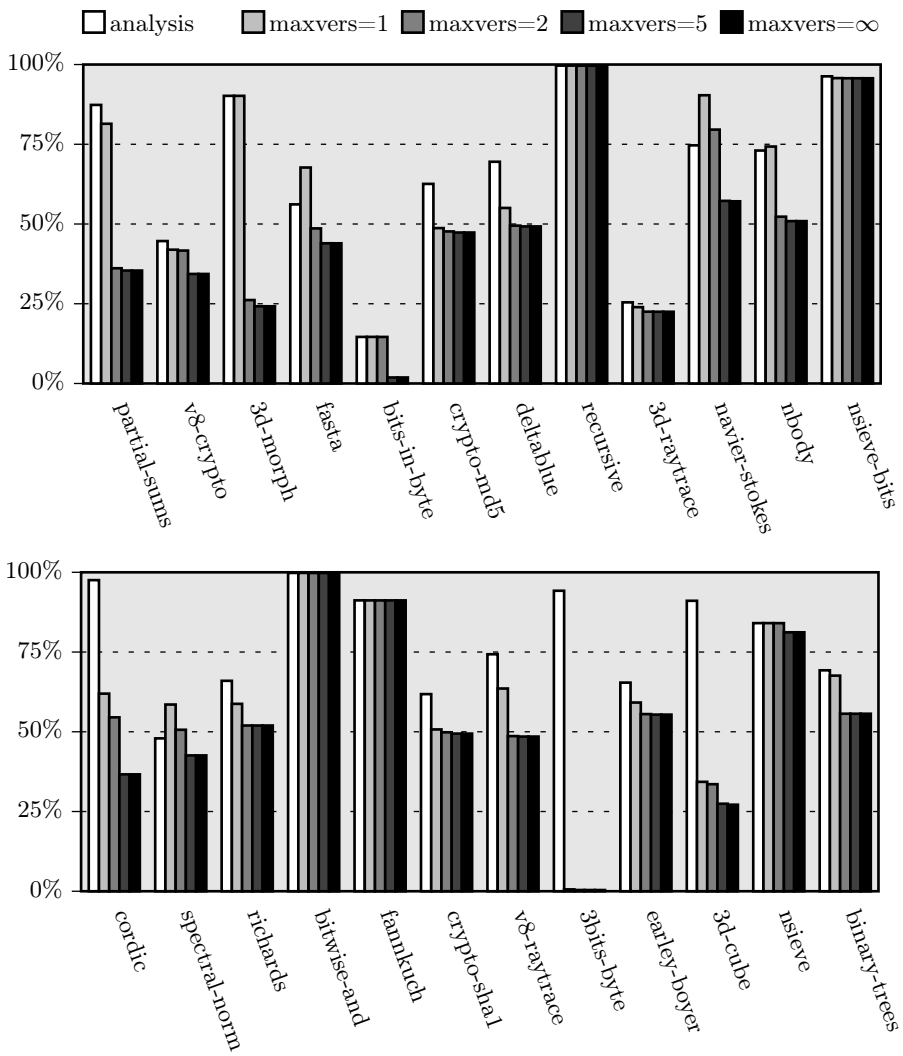

Fig. 6. Counts of dynamic type tests (relative to baseline)

```
function bitsinbyte(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}

function TimeFunc(func) {
    var x, y, t;
    for(var x = 0; x < 350; x++)
    for(var y = 0; y < 256; y++) func(y);
}
TimeFunc(bitsinbyte);
```

Fig. 7. SunSpider bits-in-byte benchmark

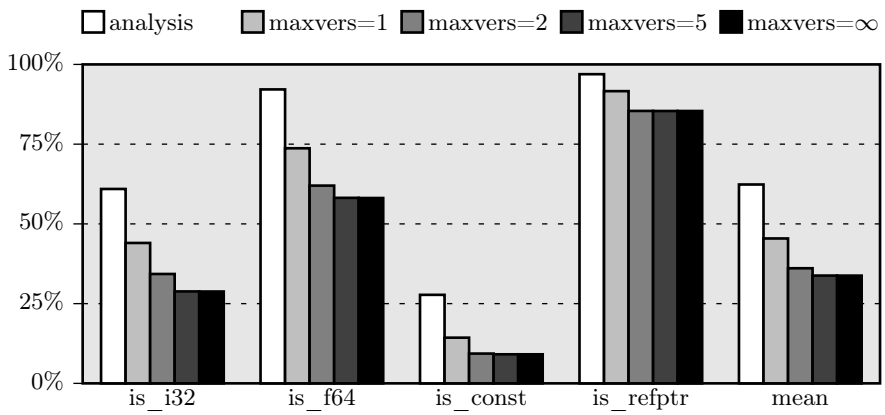

Fig. 8. Type test counts by kind of type test (relative to baseline)

ment. This is likely because property access primitives are very large, and thus seldom inlined, limiting the ability of both basic block versioning and the analysis to propagate type information for reference values. We note that versioning is much more effective than the analysis when it comes to eliminating `is_float64` type tests. This is probably because integer and floating point types often get intermixed, leading to cases where the analysis cannot eliminate such tests. The versioning approach has the advantage that it can replicate and detangle integer and floating point code paths. A limit of 5 versions per block eliminates 64% of type tests on average (geometric mean), compared to 33% for the analysis.

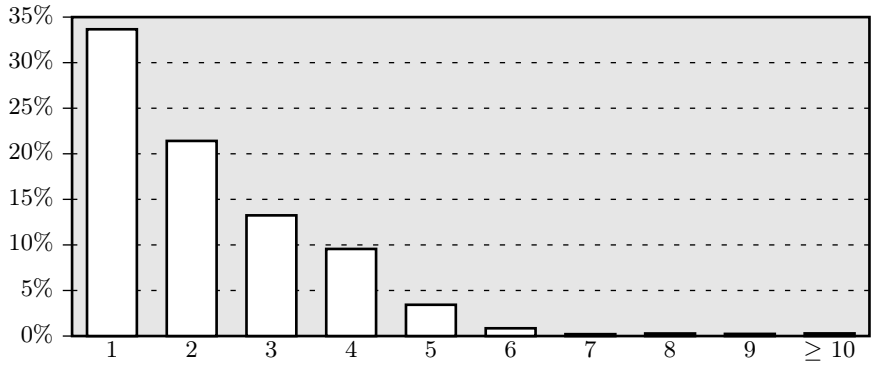

Fig. 9. Relative occurrence of block version counts

Figure 9 shows the relative proportion of blocks for which different numbers of versions were generated, averaged accross all benchmarks (geometric mean). As one might expect, the relative proportion of blocks tends to steadily decrease as the number of versions is increased. Most blocks only have one or two versions, and less than 9% have 5 versions or more. There are very few blocks which have 10 versions or more. These are a small minority, but such pathological cases do occur in practice.

The function generating the most block versions in our tests is `DrawLine` from the `3d-cube` benchmark, which produces 32 versions of one particular block. This function draws a line in screen space between point coordinates `x1,y1` and `x2,y2`. Multiple different values are computed inside `DrawLine` based on these points. Each of the coordinate values can be either integer or floating point, which results in a situation where there are several live variables, all of which can have two different types. This creates an explosion in the number of versions of blocks inside this function as basic block versioning tries to account for all possible type combinations of these values. In practice, the values are either all integer, or all floating point, but our implementation of basic block versioning is currently unable to take advantage of this helpful fact. We have experimentally verified that, in fact, only 17 of the 32 versions generated in `DrawLine` are actually executed. A strategy for addressing this problem is discussed in Section 6.

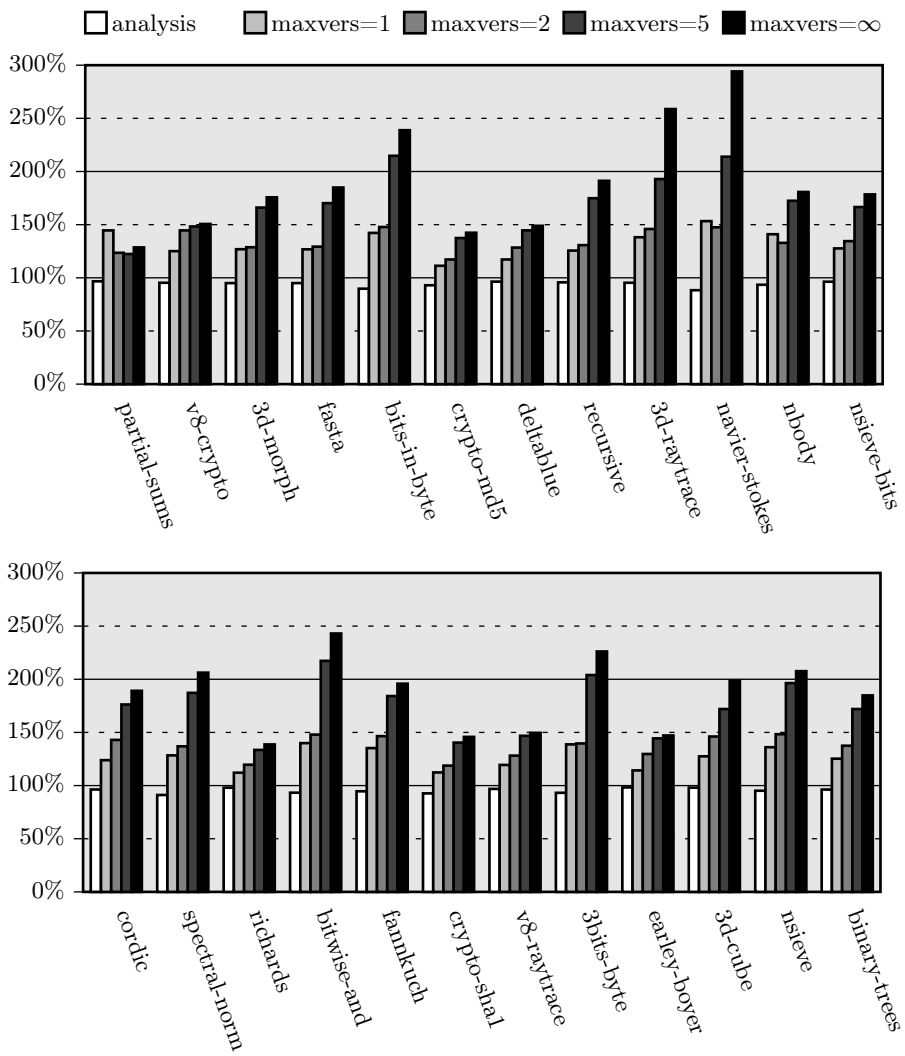

Fig. 10. Code size growth for different block version limits

The effects of basic block versioning on the total generated code size are shown in Figure 10. It is interesting to note that the representation analysis almost always results in a slight reduction in code size. This is because the analysis allows the elimination of type tests and the generation of more optimized code, which is often smaller. On the other hand, basic block versioning can generate multiple versions of basic blocks, which results in more generated code. The volume of generated code does not increase linearly with the block version cap. Rather, it tapers off as a limited number of versions tends to be generated for each block. Even without a block version limit, the code size is less than double that of the baseline in most cases. A limit of 5 versions per block results in a mean code size increase of 69%.

5 Related Work

There have been multiple efforts to devise type analyses for dynamic languages. The Rapid Atomic Type Analysis (RATA) [17] is an intraprocedural flow-sensitive analysis based on abstract interpretation which aims to assign unique types to each variable inside of a function. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [4], Ruby [10] and Python [3], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [15][16]. Such analyses are usable in the context of static code analysis, but take too long to execute to be usable in compilation and do not deal with the complexities of dynamic code loading.

More recently, work done by Brian Hackett et al. resulted in an interprocedural hybrid type analysis for JavaScript suitable for use in production JIT compilers [14]. This analysis represents a great step forward for dynamic languages, but as with other type analyses, must assign one type to each value, which makes it vulnerable to imprecise type information polluting analysis results. Basic block versioning could potentially help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Trace compilation aims to record long sequences of instructions executed inside of hot loops [12]. Such sequences of instructions often make optimization easier. Type information can be accumulated along traces and used to specialize the code to remove type tests [11], overflow checks [23] and unnecessary allocations [6]. Tracing is similar to basic block versioning, in that context updating works on essentially linear code fragments and accumulates type information during compilation. However, trace compilation incurs several difficulties and corner cases in practice, such as the potential for trace explosion if there is a large number of control-flow paths going through a loop, and poor capability to deal with code that is not loop-based. Work on trace regions by Bebenita et al. [5] introduces traces with join nodes. These join nodes can potentially elimi-

nate tail duplication among traces and avoid the problem of trace explosion, but also makes the compiler architecture more complex.

Basic block versioning bears some similarities to classic compiler optimizations such as loop unrolling [9], loop peeling [24], and tail duplication, in that it achieves some of the same results. Tail duplication and loop peeling are used in the formation of hyperblocks [18], which are sets of basic blocks grouped together, such that control-flow may enter only into one of the blocks, but may exit at multiple locations. This structure was designed to facilitate the optimization of large units of code for VLIW architectures. A parallel can be drawn between basic block versioning and Partial Redundancy Elimination (PRE) [19] in that the versioning approach seeks to eliminate and hoists out of loops a specific kind of redundant computation, that of dynamic type tests.

Basic block versioning is also similar to the idea of node splitting [25]. This technique is an analysis device designed to make control-flow graphs reducible and more amenable to analysis. The path splitting algorithm implemented in the SUIF compiler [22] aims at improving reaching definition information by replicating control-flow nodes in loops to eliminate joins. Unlike basic block versioning, these algorithm cannot gain information from type tests. The algorithms as presented are also specifically targeted at loops, while basic block versioning makes no special distinction. Similarly, a static analysis which replicates code to eliminate conditional branches has been developed [20]. This algorithm operates on a low-level intermediate representation, is intended to optimize loops and does not specifically eliminate type tests.

Customization is a technique developed to optimize the SELF programming language [7] which compiles multiple copies of methods, specialized based on the receiver object type. Similarly, type-directed cloning [21] clones methods based on argument types, which can produce more specialized code using richer type information. The work of Maxime Chevalier-Boisvert et al. on Just-In-Time (JIT) specialization for MATLAB [8] and similar work done for the MaJIC MATLAB compiler [2] tries to capture argument types to dynamically compile optimized copies of functions. All of these techniques are forms of type-driven code duplication aimed at enhancing type information. Basic block versioning operates at a lower level of granularity, which allows it to find optimization opportunities inside of method bodies by duplicating code paths.

6 Future Work

Our current implementation only tracks type information intraprocedurally. It would be desirable to extend basic block versioning in such a way that type information can cross function call boundaries. This could be accomplished by allowing functions to have multiple entry point blocks, specialized based on context information coming from callers. Similarly, call continuation blocks (return points) could also be versioned to allow information about return types to flow back into the caller.

Another obvious extension of basic block versioning would be to collect more detailed type information. For example, we may wish to propagate information about global variable types, object identity and object field types. It may also be desirable, in some cases, to know the exact value of some variable or object field, particularly if this value is likely to remain constant. Numerical range information could potentially be collected to help eliminate bound and overflow checks.

Basic block versioning, as we have implemented it, sometimes generates versions that account for type combinations that never occur in practice. This could potentially be addressed by generating stubs for the targets of cloned conditional branches. Higgs already produces stubs for unexecuted blocks, but generates all requested versions of a block if the block was ever executed in the past. Producing stubs for cloned branches would delay the generation of machine code for these branch targets until we know for a fact that they are executed, avoiding code generation for unnecessary code paths. The choice of where to generate stubs could potentially be guided by profiling data.

Some of the information accumulated and propagated by basic block versioning may not actually be useful for optimization. This is likely to become a bigger problem if the approach is extended to work accross function call boundaries, or if more precise type and constant information is accumulated. An interesting avenue may be to choose which information to propagate based on usefulness. That is, the most frequently executed type tests are probably the ones we should focus our resources on. These tests should be dynamically identified through profiling and used to decide which information to propagate.

7 Conclusion

We have introduced a novel compilation technique called context-driven basic block versioning. This technique combines code generation with type analysis to produce more optimized code through the accumulation of type information during compilation. The versioning approach is able to perform optimizations such as automatic hoisting of type tests and efficiently detangles code paths along which multiple numerical types can occur. Our experiments show that in most cases, basic block versioning eliminates significantly more dynamic type tests than is possible using a traditional flow-based type analysis. It eliminates 64% of type tests on average with a limit of 5 versions per block, compared to 33% for the analysis, and never performs worse than such an analysis.

Basic block versioning trades code size for performance. Such a tradeoff is often desirable, particularly for performance-critical application kernels. We have empirically demonstrated that although our implementation of basic block versioning does increase code size, the resulting increase is reasonably moderate, and can easily be limited with techniques as simple as a hard limit on the number of versions per basic block. In our experiments, a limit of 5 versions per block results in a mean code size increase of 69%. More sophisticated implementa-

tions that adjust the amount of code replication allowed based on the execution frequency of functions are certainly possible.

Basic block versioning is a simple and practical technique suitable for integration in real-world compilers. It requires little implementation effort and can offer important advantages in JIT-compiled environments where type analysis is often difficult. Dynamic languages, which perform a large number of dynamic type tests, stand to benefit the most.

Higgs is open source and the code used in preparing this publication is available on GitHub[2].

References

1. ECMA-262: ECMAScript Language Specification. European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edn. (2009)
2. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI). pp. 294–303. ACM New York (May 2002)
3. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the 2007 Dynamic Languages Symposium (DLS). pp. 53–64. ACM New York (2007)
4. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Proceedings of ECOOP 2005, pp. 428–452. Springer Berlin Heidelberg (2005)
5. Bebenita, M., Chang, M., Wagner, G., Gal, A., Wimmer, C., Franz, M.: Trace-based compilation in execution environments without interpreters. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ). pp. 59–68. ACM New York (2010)
6. Bolz, C.F., Cuni, A., FijaBkowski, M., Leuschel, M., Pedroni, S., Rigo, A.: Allocation removal by partial evaluation in a tracing jit. In: Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM). pp. 43–52. ACM New York (2011)
7. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In: Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI). pp. 146–160. ACM New York (Jun 1989)
8. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through just-in-time specialization. In: Proceedings of the 2010 international conference on Compiler Construction (CC). pp. 46–65. Springer Berlin Heidelberg (2010)
9. Davidson, J.W., Jinturkar, S.: Aggressive loop unrolling in a retargetable, optimizing compiler. In: Proceedings of the 1996 international conference on Compiler Construction (CC). pp. 59–73. Springer Berlin Heidelberg (1996)
10. Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static type inference for ruby. In: Proceedings of the 2009 ACM Symposium on Applied Computing (SAC). pp. 1859–1866. ACM New York (2009)

[2] https://github.com/maximecb/Higgs/tree/cc2014

11. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 2009 conference on Programming Language Design and Implementation (PLDI). pp. 465–478. ACM New York (2009)
12. Gal, A., Probst, C.W., Franz, M.: HotpathVM: an effective jit compiler for resource-constrained devices. In: Proceedings of the 2nd international conference on Virtual Execution Environments (VEE). pp. 144–153. ACM New York (2006)
13. Gudeman, D.: Representing type information in dynamically typed languages (1993)
14. Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for JavaScript. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). pp. 239–250. ACM New York (Jun 2012)
15. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Proceedings of the 16th International Symposium on Static Analysis (SAS). pp. 238–255. Springer Berlin Heidelberg (2009)
16. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: Proceedings 17th International Static Analysis Symposium (SAS). Springer Berlin Heidelberg (September 2010)
17. Logozzo, F., Venter, H.: RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In: Proceedings of the 2010 international conference on Compiler Construction (CC). pp. 66–83. Springer Berlin Heidelberg (2010)
18. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: ACM SIGMICRO Newsletter. vol. 23, pp. 45–54. IEEE Computer Society Press (1992)
19. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (Feb 1979)
20. Mueller, F., Whalley, D.B.: Avoiding conditional branches by code replication. In: Proceedings of the 1995 conference on Programming Language Design and Implementation (PLDI). pp. 56–66. ACM New York (1995)
21. Plevyak, J., Chien, A.A.: Type directed cloning for object-oriented programs. In: Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC). pp. 566–580 (1995)
22. Poletto, M.A.: Path splitting: a technique for improving data flow analysis. Ph.D. thesis, MIT Laboratory for Computer Science (1995)
23. Sol, R., Guillon, C., QuintÃčo Pereira, F., Bigonha, M.: Dynamic elimination of overflow tests in a trace compiler. In: Knoop, J. (ed.) Proceedings of the 2011 international conference on Compiler Construction (CC), pp. 2–21. Springer Berlin Heidelberg (2011)
24. Song, L., Kavi, K.M.: A technique for variable dependence driven loop peeling. In: Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on. pp. 390–395. IEEE (2002)
25. Unger, S., Mueller, F.: Handling irreducible loops: optimized node splitting versus dj-graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(4), 299–333 (Jul 2002)
26. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(2), 181–210 (Apr 1991)

A First Appendix

Algorithm 3 Type propagation analysis

```
 1: procedure typeProp(function)
 2:     outTypes ← ∅
 3:     edgeTypes ← ∅
 4:     visited ← ∅                                  ▷ Set of visited control-flow edges
 5:     workList ← {⟨null, function.entryBlock⟩}
 6:     while workList not empty do
 7:         edge ← workList.dequeue()
 8:         block ← edge.succ
 9:         if block.execCount is 0 then
10:             continue                          ▷ Ignore yet unexecuted blocks (stubs)
11:         end if
12:         visited.add(edge)
13:         curTypes ← ∅                          ▷ Merge type info from predecessors
14:         for edge in block.incoming do
15:             if edge in visited then
16:                 curTypes ← curTypes.merge(edgeTypes.get(edge))
17:             end if
18:         end for
19:         for phiNode in block.phis do
20:             t ← evalPhi(phiNode, block, visited)
21:             curTypes.set(phi, t)
22:             outTypes.set(phi, t)
23:         end for
24:         for instr in block.instrs do
25:             t ← evalInstr(instr, curTypes)
26:             curTypes.set(instr, t)
27:             outTypes.set(instr, t)
28:         end for
29:     end while
30:     return outTypes;
31: end procedure
```

Algorithm 4 Transfer functions for the type propagation analysis

```
32: procedure evalPhi(phiNode, block, visited)
33:     t ← ⊥
34:     for edge in block.incoming do
35:         if edge in visited then
36:             predType = getType(edgeTypes.get(edge), phiNode.getArg(edge))
37:             if predType is ⊥ then
38:                 return ⊥
39:             end if
40:             t = t.merge(predType)
41:         end if
42:     end for
43:     return t
44: end procedure
45: procedure AddInt32.evalInstr(instr, curTypes)
46:     return int32    ▷ The output type of AddInt32 is always int32. If an overflow occurs, the result is recomputed using AddFloat64
47: end procedure
48: procedure IsInt32.evalInstr(instr, curTypes)
49:     argType ← getType(curTypes, instr.getArg(0))
50:     if argType is ⊥ then                ▷ If the argument type is not yet evaluated
51:         return ⊥
52:     else if argType is ⊤ then                ▷ If the argument type is unknown
53:         return const
54:     else if argType is int32 then
55:         return true
56:     else
57:         return false
58:     end if
59: end procedure
60: procedure IfTrue.evalInstr(instr, curTypes)
61:     arg ← instr.getArg(0)
62:     argType ← getType(curTypes, arg)
63:     if argType is ⊥ then
64:         return ⊥
65:     end if
66:     testVal ← null
67:     testType ← ⊤
68:     if arg instanceof IsInt32 then
69:         testVal ← arg.getArg(0)    ▷ Get the SSA value whose type is being tested
70:         testType ← int32
71:     end if
72:     if argType is true or argType is const or argType is ⊤ then
73:         queueSucc(instr.trueTarget, typeMap, testVal, testType) ▷ Queue the true branch, and propagate the test value's type (if applicable)
74:     end if
75:     if argType is false or argType is const or argType is ⊤ then
76:         queueSucc(instr.falseTarget, typeMap, null, ⊤)
77:     end if
78:     return ⊥
79: end procedure
```

CHAPTER 5

LAZY BASIC BLOCK VERSIONING

5.1 A Problem with Eager BBV

In the previous chapter, we presented our first attempt at an eager BBV implementation, which generates all basic block versions for a given method at once, when this method is compiled. This was a mixed success. On the one hand, we were able to eliminate 64% of type tag tests, which is almost twice as much as what an intraprocedural dataflow analysis could achieve. However, this came at the cost of a 69% average code size increase. The large code size increase resulted in poor instruction cache utilization (see Section 8.4), which made it so we could not show measurable performance improvements in practice.

The main issue with eagerly generating block versions for a given function at method compilation time is that it requires the use of heuristics. That is, we must attempt to guess which block versions will be needed at execution time. Doing this before executing a function must inevitably rely on approximate information, which means we must conservatively generate block versions corresponding to combinations of types which will never occur at execution time.

Generating redundant block versions is problematic for multiple reasons. For one, it wastes compilation time and memory resources. It also follows that redundant versions result in inefficient use of instruction caches, since the resulting machine code must be placed somewhere, and we cannot know which versions are redundant at code generation time. Finally, redundant versions eat up part of our versioning budget, which would have been better spent on versions that will actually be executed.

The approach presented in Chapter 4 is not purely eager. Higgs segregates numerical values into small integers and floating-point values. As such, it must insert overflow checks after most integer arithmetic operations. Overflows are rare, but when they occur, integer values must be promoted to floating-point values. Unfortunately, even though

overflows are rare, it is often difficult to prove at code generation time that they cannot happen. The result is that if basic block versions are generated in a purely eager manner, many versions are wasted trying to handle cases where integer overflows could occur, but do not in practice.

The solution we have found is to avoid eagerly generating block versions for overflow paths, and instead speculate that overflows will not occur. Machine code stubs [1] are generated to detect overflows. When such a stub is executed, it calls back into the compiler, which throws away the code for the current function and recompiles it in its entirety. Recompiling whole functions is costly, but this is feasible because of how rare integer overflows are in most programs. This strategy is a form of lazy handling of integer overflows, which avoids wasting basic block versions on overflows which do not occur.

5.2 Laziness

The strategy we have used to avoid generating block versions corresponding to integer overflows is a form of speculative optimizations. Unfortunately, this strategy cannot be applied everywhere because recompiling entire functions is quite expensive. Enter lazy basic block versioning. Our key insight is that it should be possible, using machine code stubs, to lazily generate block versions when they are actually needed, that is, when a piece of code first tries to branch to the said version. Using this strategy, it then becomes possible to generate exactly the set of block versions corresponding to types that do occur at execution time, no more, no less.

Figure 5.1 illustrates a simple control-flow graph where a type test on the variable `n` branches to two blocks `A` and `B`. Given that `n` is always an `int32` at execution time, we can lazily generate versions of blocks `A` and `C`, but we do not need to generate code for block `B`. As such, we end up generating less code than we would have with eager code

1. A stub is a small piece of machine code which serves as a placeholder. Stubs are used to speculate that specific branches will not be executed. When a stub is executed, this indicates that the speculation has failed, which triggers some action on the part of the compiler. Typically, the execution of a stub results in new machine code being generated, which then takes the place of the stub.

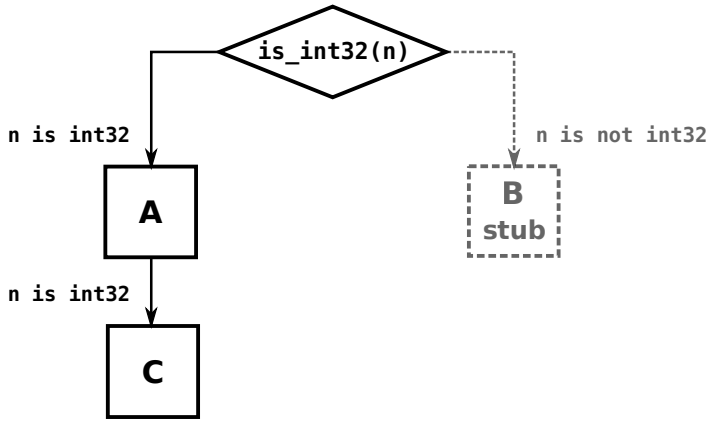

Figure 5.1: With lazy BBV, unexecuted block versions are never generated.

generation.

Lazy BBV does not compile whole methods. Instead, it generates code incrementally, one basic block at a time. This is accomplished with the use of stubs. That is, the compiler generates code for a basic block, and keeps generating code for as long as it can determine the direction of conditional branches at code generation time. When it encounters a branch for which it cannot determine which direction will be taken, stubs are generated, and execution is resumed.

Machine code for block versions is appended to an executable memory area in a linear manner, and stubs are generated in a separate executable region. Branches to stubs are dynamically rewritten when stubs are executed. In this way, machine code of good quality is generated. Blocks which execute in sequence usually follow each other in the machine code without unnecessary jump instructions being introduced.

5.3 Results

Our initial investigation into lazy BBV shows that this strategy effectively eliminates the code bloat problem associated with eager BBV. The mean code size increase is just 0.19% with lazy versioning, compared to 69% with the eager approach. Lazy versioning does not suffer from a code bloat problem because it only needs to generate code for type combinations that actually occur at execution time.

An added benefit of eliminating the code bloat problem is that we do not waste our versioning budget on unnecessary versions. Instead, we can spend this budget on block versions that actually occur. As a result, we were able to eliminate 71% of dynamic type tests on average, instead of 64% with eager BBV.

Furthermore, the lazy BBV approach can help in generating higher quality machine code in a third way. Because block versions are generated lazily, they tend to be produced in the order that they are first executed. This strategy is an effective way, in practice, of ordering executable code in memory. It tends to generate linear sequences of block versions that directly fall through to one another without branching.

With lazy versioning, we were able to obtain average speedups of 21% over a baseline without BBV, demonstrating that the technique does yield performance improvements in practice. The article which follows was submitted and published at the ECOOP 2015 conference [7].

Simple and Effective Type Check Removal through Lazy Basic Block Versioning

Maxime Chevalier-Boisvert[1] and Marc Feeley[2]

1 **DIRO, Université de Montréal**
Montréal, QC, Canada
2 **DIRO, Université de Montréal**
Montréal, QC, Canada

Abstract

Dynamically typed programming languages such as JavaScript and Python defer type checking to run time. In order to maximize performance, dynamic language VM implementations must attempt to eliminate redundant dynamic type checks. However, type inference analyses are often costly and involve tradeoffs between compilation time and resulting precision. This has lead to the creation of increasingly complex multi-tiered VM architectures.

This paper introduces *lazy basic block versioning*, a simple JIT compilation technique which effectively removes redundant type checks from critical code paths. This novel approach lazily generates type-specialized versions of basic blocks on the fly while propagating context-dependent type information. This does not require the use of costly program analyses, is not restricted by the precision limitations of traditional type analyses and avoids the implementation complexity of speculative optimization techniques.

We have implemented intraprocedural lazy basic block versioning in a JavaScript JIT compiler. This approach is compared with a classical flow-based type analysis. Lazy basic block versioning performs as well or better on all benchmarks. On average, 71% of type tests are eliminated, yielding speedups of up to 50%. We also show that our implementation generates more efficient machine code than TraceMonkey, a tracing JIT compiler for JavaScript, on several benchmarks. The combination of implementation simplicity, low algorithmic complexity and good run time performance makes basic block versioning attractive for baseline JIT compilers.

1998 ACM Subject Classification D.3.4 – compilers, optimization, code generation, run-time environments

Keywords and phrases Just-In-Time Compilation, Dynamic Optimization, Type Checking, Code Generation, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

A central feature of dynamic programming languages is that they defer type checking to run time. In order to maximize performance, efficient implementations of dynamic languages seek to type-specialize code so as to eliminate dynamic type checks when possible. Doing so requires proving that these type checks are unnecessary and generating type-specialized code.

Traditionally, the main approach for eliminating type checks has been to use type inference analyses. This is problematic for modern dynamic languages such as JavaScript and Python for three main reasons. The first is that these languages are generally poorly amenable to whole-program type analyses. Constructs such as `eval` and dynamic loading of modules can destroy previously valid type information. The second is that these analyses can be

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1022

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

expensive in terms of computation time and memory usage, making them unsuitable for JIT compilers, particularly baseline compilers. To reduce analysis cost, it is often necessary to sacrifice precision. A last issue is that some type checks simply cannot be eliminated through analysis alone, without code transformations.

Because dynamic programming languages are generally poorly amenable to type inference, and whole-program analyses are often too expensive for JIT compilation purposes, state of the art JavaScript VMs such as SpiderMonkey, V8 and JavaScriptCore rely on increasingly complex multi-tiered architectures integrating interpreters and multiple JIT compilers with different optimization capabilities (baseline compilers to aggressively optimizing compilers). At the highest optimization levels, modern JIT compilers typically make use of type feedback, type inference analysis and also speculative optimization and deoptimization [16] with On-Stack Replacement (OSR).

We introduce a simple approach to JIT compilation that generates efficient type-specialized code without the use of costly type inference analyses or type profiling. The approach, which we call lazy basic block versioning, lazily clones and specializes basic blocks on the fly in a way that allows the compiler to accumulate type information while machine code is generated, without a separate type analysis pass. The accumulated information allows the removal of redundant type tests, particularly in performance-critical paths.

Lazy basic block versioning lets the execution of the program itself drive the generation of type-specialized code, and is able to avoid some of the precision limitations of traditional, conservative type analyses as well as avoiding the implementation complexity of speculative optimization techniques.

This paper relates our experience implementing lazy basic block versioning and reports on its effectiveness as a code generation technique. The rest of the paper is organized as follows. Section 2 explains the basic block versioning approach, comparing it with the related approaches of static type analysis and trace compilation. Section 3 describes an implementation within Higgs, an experimental JIT compiler for JavaScript. Section 4 presents an evaluation of the performance of this implementation. Related work is presented in Section 5.

2 Basic Block Versioning

In the basic block versioning approach, the code generator maintains a typing context (or type map) which indicates what is known of the type of each live local variable at the current program point. All local variables start out with the *unknown* type at function entry points. While generating code, the code generator updates the typing context by inferring the result type of data operations it encounters. Conditional branch instructions corresponding to type tests create two new typing contexts for outgoing branch edges. In each context, a type is assigned to the variable being tested (either the type tested or *unknown*). When a type test branch instruction is encountered and the type of the argument is known, the branch direction can be determined at code generation time and the type test eliminated.

The compiler may generate code for multiple instances of a given basic block; one version for each typing context encountered on a branch to that basic block. This allows specializing the basic block and its successors by taking the types of live variables into account. While basic block versioning works at the level of individual basic blocks, the propagation of typing contexts to successor blocks allows type-specializing entire control flow graphs.

An important difference between this approach and traditional type analyses is that basic block versioning does not compute a fixed point on types to be inferred. Variables which

may have multiple different types at the same program point are handled more precisely with basic block versioning due to the duplication of code. In a traditional type analysis, the union of several possible types would be assigned to such variables, causing the analysis to be conservative. With basic block versioning, distinct basic block versions, and thus distinct code paths, will be created for each type previously encountered, allowing a precise context-dependent tracking of types.

With basic block versioning, loops in the control flow graph need not be handled specially. A first version of the loop header is generated for a typing context C_1. At the point(s) where control flow branches back to the loop header, a new version of the loop may be generated if the typing context C_2 is different from C_1. Given that the number of possible contexts is finite, a fixed point is eventually reached, that is, the typing context at branches to the loop header will eventually match one of C_1, C_2, ..., C_N. The number of versions actually generated is expected to be low because the type of most variables remains stable for the duration of a function.

There is a risk of a combinatorial explosion when multiple versions of basic blocks are created eagerly. Consider the simple statement `x=a+b+c+d`. If the types of `a`, `b`, `c` and `d` are *unknown*, and those variables are live after the assignment, and there are two possible numerical types (`int` and `float`), there could be up to 16 versions of the basic block containing the assignment to `x`, one version for each set of type assignments to the variables being summed. In general, if basic block versioning is performed in an *eager* fashion and there are t possible types of values and a function has v variables, then there can be up to t^v versions of some basic blocks in that function. However, the logic of a program puts constraints on possible type combinations. In practice, not all the combinations of types are observed during an execution of a program.

It is often the case that variables are monomorphic in type (i.e. they always contain the same type of value). We can take advantage of this by *lazily* creating new block versions on demand. Versions for a particular context are only generated when that context is encountered during execution. *Lazy basic block versioning* doesn't completely eliminate the possibility of a combinatorial explosion in pathological cases, but this can be prevented by placing a hard limit on the number of versions generated for any given block. Some increase in code size is to be expected, but no more than a constant factor. Mueller and Whalley have shown [24] that specializing code through replication, while increasing the static size of machine code, can reduce the dynamic count of executed instructions and result in better cache usage.

Traditional type analyses often cannot infer a type for a variable, either because there is insufficient semantic information in the source program, or because the analysis is limited in its capabilities. For example, with an intraprocedural type analysis of JavaScript, no type information is known about function parameters. Without transforming the program, many variable types cannot be recovered by analysis alone. Moreover, the *unknown* type may propagate through primitive operations and effectively poison the results of such type analyses.

As will be demonstrated in Section 4, a key advantage of basic block versioning over program analyses lies in its ability to *recover unknown types*. The versioning approach is able to exploit type tests that are implicitly part of the language semantics to gain type information, and then generate new block versions where the additional type information remains known. Basic block versioning automatically unrolls some of the first iterations of loops in such a way that type tests are hoisted out of loop bodies. For example, if variables of *unknown* type are used unconditionally in a loop, their type will be tested only in the

first iteration of the loop. The type information gained will allow further iterations to avoid redundant type tests.

Lazy basic block versioning bears some similarity to trace compilation [5] in the use of code duplication and type specialization to eliminate type tests [13]. Trace compilation typically relies on an interpreter to detect hot loops and record traces. It is also most effective on loop-heavy code. In contrast, lazy basic block versioning can handle any code structure just as effectively. It avoids the dual language implementation (interpreter and trace compiler) and requires no special infrastructure for profiling or recording traces.

The relative simplicity of tracking typing contexts and previously generated basic block versions means that the compiler avoids algorithms of high computational complexity. With a hard limit on the number of block versions, code generation time and code size scale linearly with the size of the input program. Lazy basic block versioning requires no external optimization or analysis passes to generate type-specialized code. This makes the approach interesting for use in baseline JIT compilers.

3 Implementation in Higgs

We have implemented lazy basic block versioning inside a JavaScript virtual machine called Higgs. This virtual machine comprises a JIT compiler targeted at x86-64 POSIX platforms. The current implementation of Higgs supports most of the ECMAScript 5 specification [18], with the exception of the `with` statement and the limitation that `eval` can only access global variables, not locals. Its runtime and standard libraries are self-hosted, written in an extended dialect of ECMAScript with low-level primitives. These low-level primitives are special instructions which allow expressing type tests as well as integer and floating point machine instructions in the source language.

In Higgs, functions are parsed into an abstract syntax tree and lazily compiled to a Static Single Assignment (SSA) Intermediate Representation (IR) when they are first called. Inlining is performed at this time according to simple fixed heuristic rules. Specific JavaScript runtime functions including arithmetic, comparison and object property access primitives are always inlined. This inlining allows exposing type tests and typed low-level operations contained inside primitives to the backend, which implements basic block versioning.

A basic block version corresponds to a basic block and an associated context containing type information about live values at the start of the block. Machine code generation always begins with the function's entry block and a default entry context being queued for compilation. Typing contexts in Higgs are implemented as sets of pairs associating live SSA values to unique type tags (see Section 3.2). Values for which no type information is known do not appear in the set. As each instruction in a block is compiled, information is both retrieved from and inserted into the current context. Information retrieved may be used to optimize the compilation of the current instruction (e.g. eliminate type tests). Instructions will also write their own output type in the context if known.

To guard against pathological cases where an unreasonably large number of versions would be generated, we have added one tunable parameter, `maxvers`, which specifies the maximum number of specialized versions that can be generated for any given basic block. Before the limit for a given block is reached, requests for new versions matching an incoming context will either find an existing exact match for the context, or compile a new version matching the incoming context exactly. Once the limit is reached for a particular block, requests for new versions of this block will first try to find an inexact but compatible match for the incoming context. An existing version is compatible with the incoming context if the

```
/**
Context compatibility test function:
- Perfectly matching contexts produce score 0
- Imperfect matches produce a score > 0
- Incompatible matches produce Infinity
*/
Number contextComp(Context predCtx, Context succCtx)
{
    Number score = 0;

    // For each value live in the successor
    foreach (value in succCtx)
    {
        auto predType = predCtx.getType(value);
        auto succType = succCtx.getType(value);

        // If the successor has no known type,
        // we would lose a known type
        if (predType != UNKNOWN &&
            succType == UNKNOWN)
            score += 1

        // If the types do not match,
        // contexts are incompatible
        else if (predType != succType)
            return Infinity;
    }

    return score;
}
```

Figure 1 Context compatibility test function

value types assumed by the existing version are a subset of those specified in the incoming context.

The context compatibility test is shown in Figure 1. A context containing less constraining types than the incoming context is compatible, but one that has more constraining types than the incoming context is not. Essentially, this allows for the loss of type information when transitioning along control flow edges. If the version limit was reached and no compatible match is found for a given block, a fully generic version of the block that assigns the *unknown* type to all live variables will be generated. This generic version is compatible with all possible incoming contexts. When the `maxvers` parameter is set to zero, basic block versioning is disabled, and only one generic version of each basic block may be generated.

3.1 Lazy Code Generation

Limiting the number of versions generated by *eager* basic block versioning to avoid combinatorial code growth is a difficult problem. Simply imposing a hard version limit is not a satisfactory solution because it is nontrivial to determine ahead of time which typing contexts are more probable than others, and which may not occur at all. This is particularly problematic in a JIT compiler, since compiling versions for type combinations that will not occur at run time translates into wasted compilation time, code bloat and poor usage of the instruction cache. There is also the issue of ordering machine code in memory so as to minimize the number of branches taken.

Clearly, basic block versioning ought to be guided by run time types, but gathering profiling data using traditional means could be expensive. Furthermore, the resulting data

may be large and complex to analyze. Instead, Higgs delays the generation of block versions and lets the run time behavior of programs drive this process. The *execution of conditional branches* triggers the generation of new block versions. This is particularly useful since all type tests are conditional branches. Versions are generated according to the types that actually occur at run time. This *lazy code generation* approach has four key benefits:

1. The order in which versions for different type combinations are generated tends to approximate the frequency of occurrence of the said types. This is particularly helpful in the presence of a block version limit.
2. It tends to produce efficient, cache-friendly linear orderings of compiled blocks in memory, as versions are generated in the order they are first executed.
3. Neither memory nor time are wasted compiling block versions for type combinations that never occur at run time. Type combinations that do not occur are never accounted for.
4. Unexecuted blocks are never compiled. Exception handling code is not generated for programs which do not throw exceptions. Floating point code is not generated for programs which do not make use of floating point values.

The Higgs backend lazily compiles versions of individual SSA basic blocks into x86-64 machine code as they are first executed. Higgs does not compile whole functions at once. Instead, the JIT compilation model employed by Higgs interleaves execution and compilation of basic blocks. The last instruction of a block, which must be a branch instruction, determines which block will be compiled next. If the branch is unconditional, or if its direction can be determined at compilation time, no branch instruction is generated, and the successor version the branch leads to is immediately compiled (unless already compiled, in which case a direct jump is written instead).

When a conditional branch whose direction cannot be determined at compilation time is encountered, a pair of out-of-line stubs are generated for the two possible outcomes of the branch, and execution resumes. Stubs, when executed, call back the compiler requesting compilation of the corresponding destination basic block with the typing context at the branch. The branch is then overwritten to fall through or jump to the generated basic block version. This way, the compilation of a particular basic block version is delayed until it is required for execution.

3.2 Type Tags and Runtime Primitives

Higgs segregates values into a few categories based on type tags [15]. These categories are: 32-bit integers (`int32`), 64-bit floating point values (`float64`), miscellaneous JavaScript constants (`const`), and four kinds of garbage-collected pointers inside the heap (`string`, `object`, `array`, `closure`). These type tags form a simple, first-degree notion of types that is used to drive the basic block versioning approach.

We chose this coarse-grained type classification to investigate the effectiveness and potential of basic block versioning. Higgs implements JavaScript operators as runtime library functions written in an extended dialect of JavaScript, and most of these functions use type tags to do type dispatching. As such, eliminating this first level of type tests as well as boxing and unboxing overhead, is crucial to improving the performance of the system as a whole.

Figure 2 illustrates the implementation of the `+` operator as an example. As can be seen, this function makes extensive use of low-level type test primitives such as `is_i32` and `is_f64` to implement dynamic dispatch based on the type tags of the input arguments. Most

```
function add(x, y) {
    if (is_i32(x)) { // If x is integer
        if (is_i32(y)) {
            if (var r = add_i32_ovf(x, y))
                return r;
            else // Handle the overflow case
                return add_f64(i32_to_f64(x),
                               i32_to_f64(y));
        } else if (is_f64(y))
            return add_f64(i32_to_f64(x), y);
    } else if (is_f64(x)) { // If x is fp
        if (is_i32(y))
            return add_f64(x, i32_to_f64(y));
        else if (is_f64(y))
            return add_f64(x, y);
    }

    // Eval args as strings and concat them
    return strcat(toString(x), toString(y));
}
```

Figure 2 Implementation of the + operator

other arithmetic, comparison and property access primitives implement a similar dispatch mechanism.

Note that while according to the ES5 specification all JavaScript numbers are IEEE double-precision floating point values, high-performance JavaScript VMs typically attempt to represent small integer values using machine integers so as to improve performance by using lower latency integer arithmetic instructions. We have made the same design choice for Higgs. Consequently, JavaScript numbers are represented using tagged `int32` or `float64` values. Arithmetic operations on `int32` values may yield an `int32` or `float64` result, but arithmetic operations on `float64` values always yield an `float64` result.

3.3 Flow-based Representation Analysis

To provide a point of comparison and contrast the capabilities of basic block versioning with that of more traditional type analysis approaches, we have implemented a forward flow-based representation analysis that computes a fixed point on the types of SSA values. The analysis is an adaptation of Wegbreit's algorithm as described in [31]. It is an intraprocedural constant propagation analysis that propagates the types of SSA values in a flow-sensitive manner.

The representation analysis uses sets of possible type tags as a type representation. It is able to gain information from typed primitives (e.g. `add_f64` produces `float64` values) as well as type tests and forward this information along branches. The analysis is also able to deduce, in some cases, that specific branches will not be executed and ignore the effects of code that was determined dead. The type tags are the same as those used by basic block versioning, with the difference that basic block versioning only propagates unique known types and not type sets (e.g. `int32` $\cup$ `float64`). This means that basic block versioning can only propagate positive information gained from type tests whereas the analysis can propagate both positive and negative information (e.g. `a` is not `int32`).

We have chosen to give the type analysis a richer type representation than that of basic block versioning because several common arithmetic primitives can produce overflows that cannot be statically predicted. This means that most arithmetic operations can produce either `int32` or `float64` types. If the type analysis could not represent this type set, it would

be forced to infer that the output type of most arithmetic operations is of *unknown* type. This would immediately put the type analysis at an enormous disadvantage when compared to basic block versioning because basic block versioning is not affected by overflows that do not occur at run time.

3.4 Concrete Example

```
function sum(n) {
    for (var i=0, s=0; i<n; i++)
        s += i;
    return s;
}
```

Figure 3 The `sum` function.

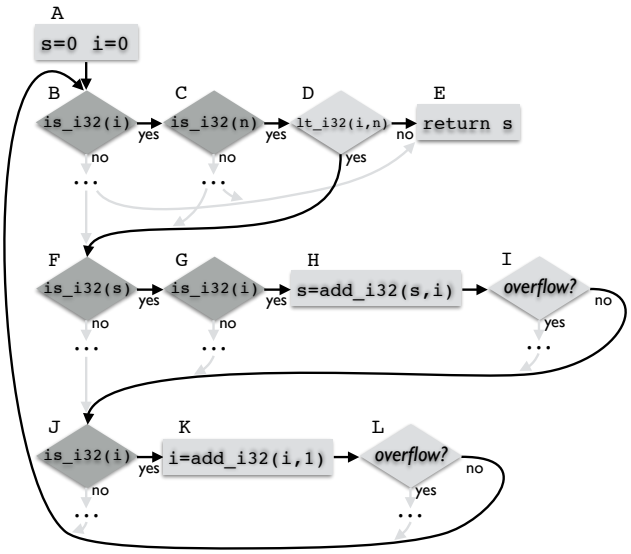

Figure 4 Control flow graph of `sum` function (unexecuted parts omitted).

To illustrate the lazy basic block versioning approach, we will explain the compilation of the `sum` function given in Figure 3. Specifically, we trace the execution of the function call `sum(500)`. This only requires 32-bit integer computations because no overflows occur for this value of `n`. Figure 4 shows the parts of the control flow graph of the function executed during this call. The complete graph is larger and includes code to handle floating point values and other types. Unexecuted parts of the control flow graph are shown as ellipses (...).

Before versioning, there are 5 type tests on `i` and `n` executed as part of the loop. Higgs compiles the code for the `sum` function incrementally, type-specializing and eliminating type tests as compilation proceeds. The compiled and specialized code is equivalent to the control flow graph shown in Figure 5. Multiple blocks have been specialized based on the knowledge that `i`, `s` and `n` are of the `int32` type. Only one type test is left, in block `C1`, and this type test has been hoisted out of the loop. It is executed only once per call to `sum`.

The incremental compilation process occurs in six steps and is illustrated in Figure 6. When first entering the `sum` function, a version of the entry block `A` is compiled, generating

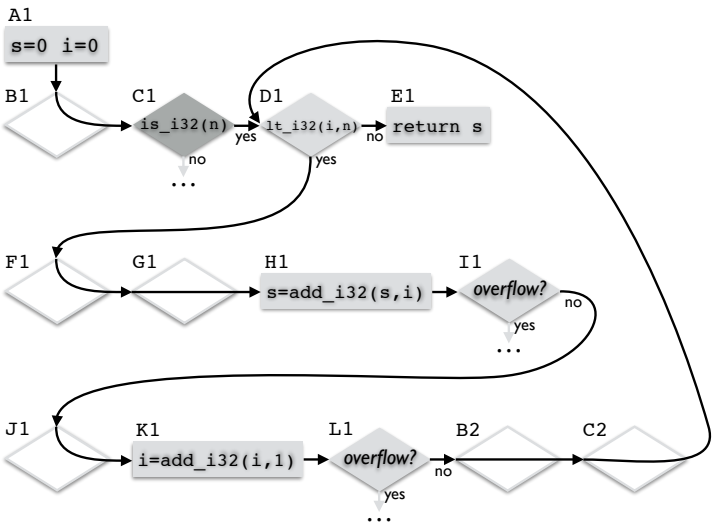

Figure 5 Control flow graph of `sum` function transformed by basic block versioning.

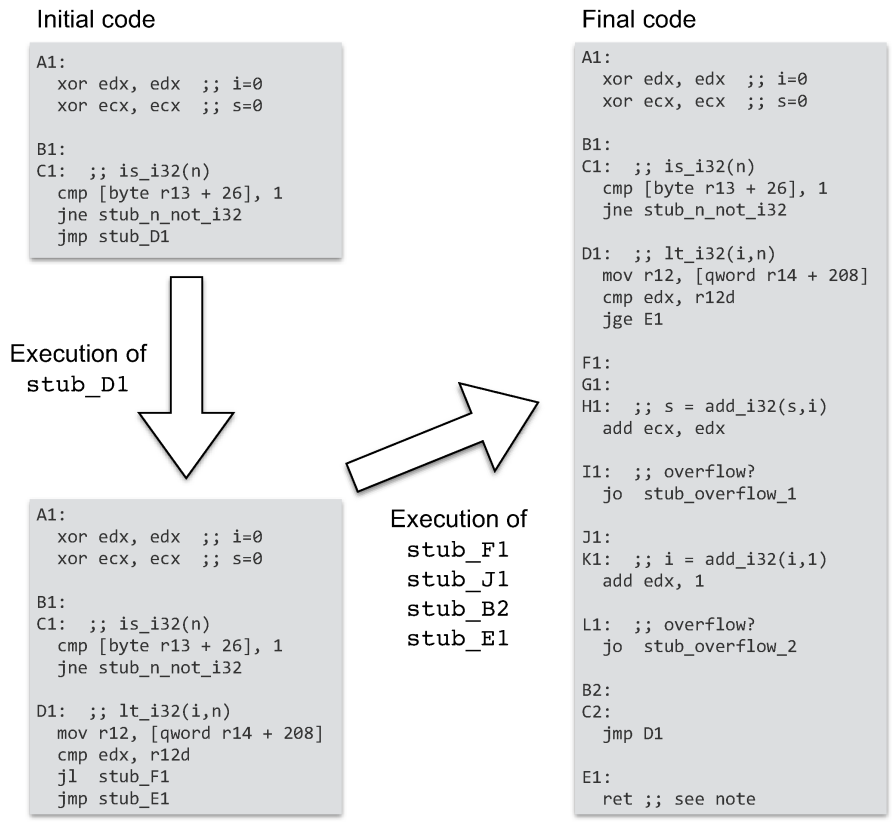

Figure 6 Machine code at different steps of code compilation

A1. Variables s and i are initialized to int32 and this is noted in the current typing context. Then, block B is compiled down to nothing because i is known to be int32 in the current context. In C1, generated from block C, the type test on n needs to emit machine code because the type of n is *unknown* in the current context and so must be tested. Therefore, stubs stub_n_not_i32 and stub_D1 are generated and execution resumes at A1.

Because n contains an int32, execution flows to stub_D1, which calls back into the JIT compiler. The branch instructions at the end of block C1 is rewritten so that a jump to a stub is executed only if n is not int32. In future calls of sum where n is int32, the branch will fall through to block D1. The generation of block D1 from D is handled similarly. Two stubs (stub_F1 and stub_E1) are used to determine the direction of the less-than comparison branch, which is unknown at compilation time. Execution then resumes at D1 and flows to stub_F1. This time, the JIT compiler inverts the direction of the branches at the end of block D1 so that the fall through will be block F1. Then blocks F1, G1, H1, and I1 are generated and execution resumes at F1.

The code is incrementally generated in this fashion by successively executing stub_J1, stub_B2, and stub_E1. After the execution of stub_B2, the emitted code executes until the end of the loop. In the last loop iteration, the less-than comparison in D1 fails. This triggers compilation of the loop exit block E1, which is conveniently placed outside of the loop body. We note that the detailed sequence of instructions needed to return from sum is more complex than what is shown (to support JavaScript's variable arity function calls).

The right part of Figure 6 shows the generated code after the execution of sum(500) has completed. Type tests in blocks F1, G1 and J1 were eliminated because i, s and n are known to be int32 at those points. The jump back to the loop header in L1 generated new versions of blocks B and C where i, s and n are known to be int32. Hence, only the first loop iteration performs a type test.

4 Evaluation

4.1 Experimental Setup

To assess the effectiveness of basic block versioning, we have tested it on a total of 26 classic benchmarks from the SunSpider and Google V8 suites. One benchmark from the SunSpider suite and one from the V8 suite were not included in our tests because Higgs does not yet implement the required features. Benchmarks making use of regular expressions were discarded because unlike V8 and TraceMonkey, Higgs does not implement JIT compilation of regular expressions, and neither does Truffle JS [33, 32].

Since Higgs interleaves compilation and execution and which parts of a program are eventually compiled is entirely dependent on run time behavior, we have measured approximate compilation times using a microsecond counter which is started and stopped when compilation begins and ends. The total times accumulated are averaged across 10 runs to give a final compilation time figure.

To measure execution time separately from compilation time in a manner compatible with V8, TraceMonkey, Truffle JS and Higgs, we have modified benchmarks so that they could be run in a loop. A number of warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place.

The number of warmup and timing iterations were scaled so that short-running benchmarks would execute for at least 1000ms in total during both warmup and timing. Unless otherwise specified, all benchmarks were run for at least 10 warmup iterations and 10 timing iterations.

V8 version 3.29.66, TraceMonkey version 1.8.5+ and Truffle JS v0.5 were used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 quad-core CPU with 8MB L3 cache and 16GB of RAM running Ubuntu Linux 12.04. Dynamic CPU frequency scaling was disabled for our experiments.

4.2 Dynamic Type Tests

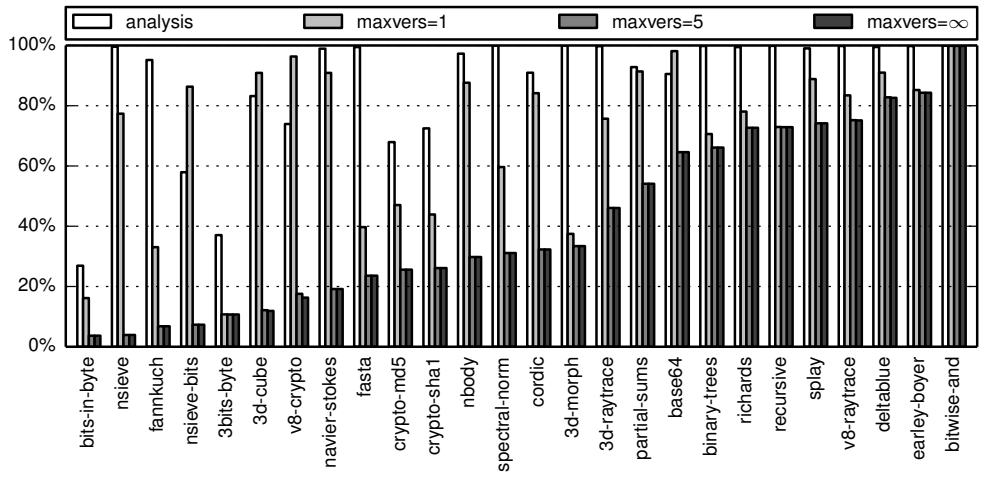

Figure 7 Dynamic counts of type tests executed using the representation analysis and lazy basic block versioning with various version limits (relative to baseline)

Figure 7 shows the dynamic counts of type tests for the representation analysis and for lazy basic block versioning with various block version limits. These counts are relative to a baseline which has the version limit set to 0, and thus only generates a generic version of each basic block. As can be seen from the counts, the analysis produces a reduction in the number of dynamically executed type tests over the unoptimized baseline on most benchmarks. The basic block versioning approach does at least as well as the analysis, and almost always significantly better. Surprisingly, even with a version cap as low as 1 version per basic block, the versioning approach is often competitive with the representation analysis. This is likely because most value types are monomorphic.

Raising the version cap reduces the number of type tests performed with the versioning approach in an asymptotic manner as we get closer to the limit of what is achievable with our implementation. The versioning approach does quite well on the `bits-in-byte` benchmark. This benchmark (see Figure 8) is an ideal use case for our versioning approach. It is a tight loop performing bitwise and arithmetic operations on integers which are all stored in local variables. The versioning approach performs noticeably better than the analysis on this test because it is able to test the type of the function parameter `b`, which is initially unknown when entering `bitsinbyte` only once per function call and propagate this type thereafter. The analysis on its own cannot achieve this, and so must repeat the test for each operation on `b`. In contrast, the `bitwise-and` benchmark operates exclusively on global variables, for which our system cannot extract types, and so neither the type analysis nor the versioning approach show any improvement over baseline for this benchmark.

A breakdown of relative type test counts by kind, averaged across all benchmarks (using

```
function bitsinbyte(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}

function TimeFunc(func) {
    var x, y, t;
    for(var x=0; x<350; x++)
        for(var y=0; y<256; y++) func(y);
}

TimeFunc(bitsinbyte);
```

Figure 8 SunSpider bits-in-byte benchmark

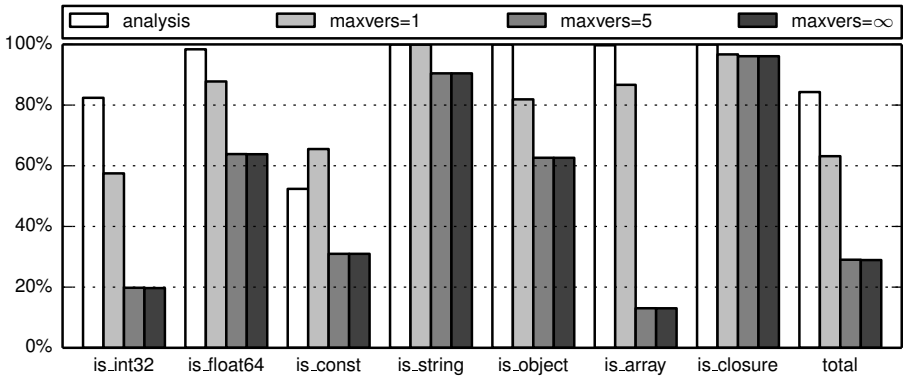

Figure 9 Type test counts by kind of type test (relative to baseline)

the geometric mean) is shown in Figure 9. We see that the versioning approach is able to perform as well or better than the representation analysis across each kind of type test. The `is_closure` category shows the least improvement. This is because functions are typically globals or methods, which basic block versioning cannot yet get type information about. We note that versioning is much more effective than the analysis when it comes to eliminating `is_i32` type tests. This is because integer and floating point types often get intermixed, leading to cases where the analysis cannot eliminate such tests. The versioning approach has the advantage that it can replicate and detangle integer and floating point code paths. A limit of 5 versions per block eliminates 71% of total type tests, compared to 16% for the analysis.

4.3 Code Size Growth

Figure 10 shows the relative proportion of blocks for which different counts of versions were generated across all benchmarks. As one might expect, the relative proportion of blocks tends to steadily decrease as the number of versions is increased. Most basic blocks have only one version, 5.2% have two, and just 0.16% of blocks have 5 versions or more. Hence, blocks with a large number of versions are a rare occurrence.

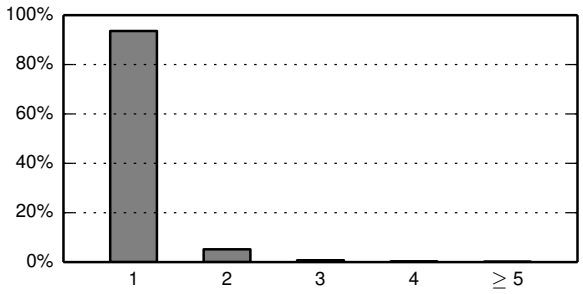

Figure 10 Relative occurrence of block version counts

The maximum number of versions ever produced for a given block across our benchmarks is 11. This occurs in the `v8-raytrace` benchmark. The function generating the most block versions in this benchmark is `rayTrace`. This function is at the core of the ray tracing algorithm. It contains a loop with several live variables used during iteration. Some of these variables can be either `null` or an object reference. There are also versions generated where basic block versioning cannot determine a type for some variables.

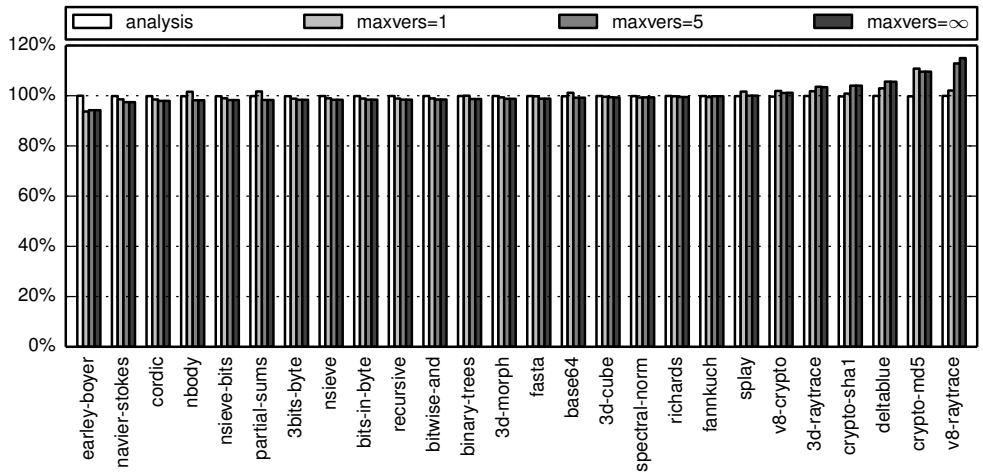

Figure 11 Code size for various block version limits (relative to baseline)

The effects of basic block versioning on the total generated code size are shown in Figure 11. It is interesting to note that the representation analysis almost always results in a slight reduction in code size. This is because the analysis allows the elimination of type tests and the generation of more optimized code, which is usually smaller. On the other hand, basic block versioning can generate multiple versions of basic blocks, which often (but not always) results in more generated code. The volume of generated code does not increase linearly with the block version limit. Rather, it tapers off as a limited number of versions tends to be generated for each block. A limit of 5 versions per block results in a mean code size increase of 0.19%. With no limit at all on the number of versions, the code size increase does not change much, with a mean of 0.25% and a maximum increase of 15% across all benchmarks. On the benchmarks we have tested, there is no pathological code size explosion, and the

block version limit is not strictly necessary.

4.4 Execution Time

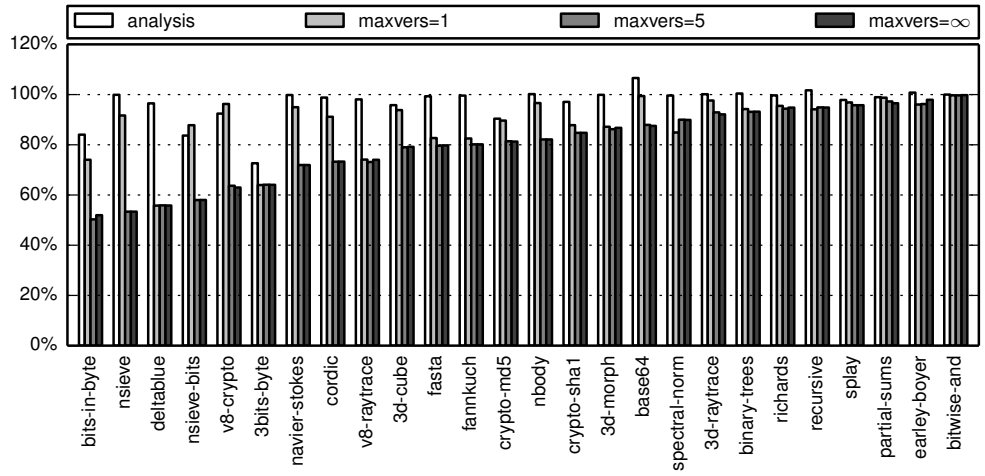

Figure 12 Execution time for various block version limits (relative to baseline)

Figure 12 shows the execution times relative to baseline. Because our type analysis is not optimized for speed and incurs a significant compilation time penalty, we have excluded compilation time and measured only time spent executing compiled machine code. A limit of 5 versions per block produces on average a 21% reduction in execution time, and speedups of up to 50%, while the type analysis yields a 4% average speedup.

In most cases, basic block versioning produces a notable reduction in relative execution time that compares favorably with the static analysis. The intraprocedural type analysis does not eliminate enough type tests to be effective in improving execution times. We believe that it should be possible to significantly improve upon the basic block versioning results with method inlining and better optimized property accesses, which would expose more type tests and more precise type information.

4.5 Eager Versioning

In order to evaluate the importance of lazyness in our basic block versioning approach, we have tested an older version of Higgs which generates block versions eagerly. In this configuration, whole methods are compiled at once, never producing stubs, and specialized versions are generated for a given block until the block version limit is hit. The versions are generated in no particular order. The performance obtained with eager generation of block versions was found to be inferior on all metrics. When the version limit is set to 5, on average, the eager approach eliminates about half as many type tests as the lazy approach, the code size is 223% of baseline on average (see Figure 13), and the execution time is 5% slower than baseline.

There are multiple issues with the eager generation of block versions. The most important one is that without some form of lazyness, without code stubs, we must always produce code for both sides of a conditional branch. In the case of eager basic block versioning, this

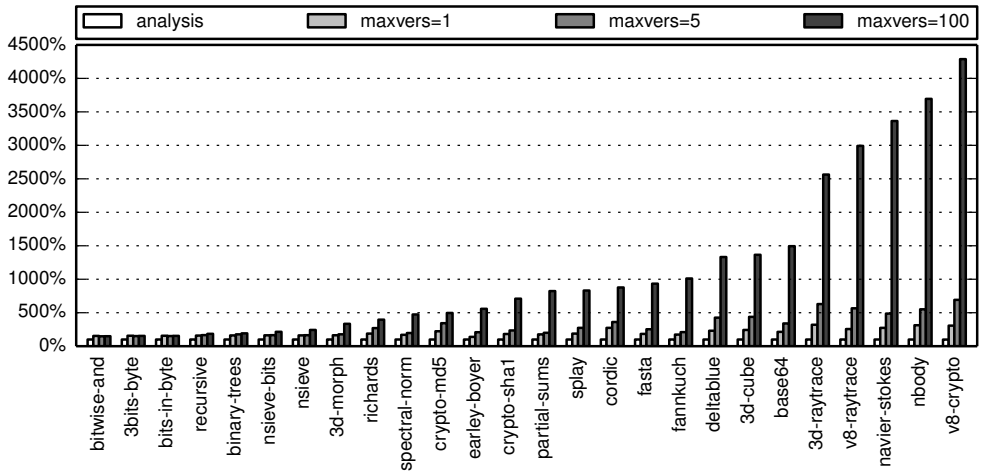

Figure 13 Code size with eager basic block versioning (relative to baseline)

means we generate code for both branches of a type test, even though in most cases only one side of the branch is ever taken. We end up generating versions for a large number of type combinations which cannot occur at run time, but which we have no heuristic to discard at method compilation time. The number of possible type combinations increases exponentially with the number of live variables, and so the block version limit is rapidly reached. Since versions are generated in no particular order, the specialized versions eagerly generated before the block version limit is hit are likely to be versions matching irrelevant type combinations.

4.6 Compilation Time

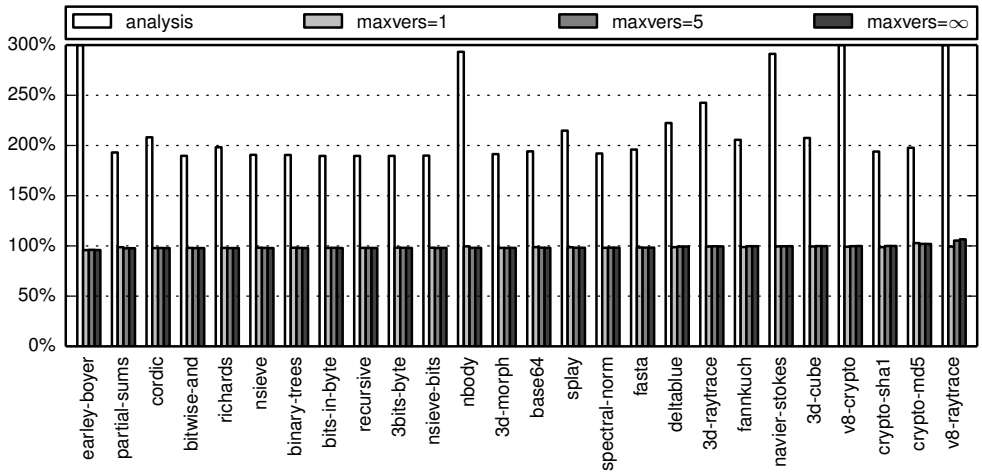

Figure 14 Compilation time for various block version limits (relative to baseline)

The graph in Figure 14 shows a comparison of the total compilation time with the type

analysis and with different block version limits relative to baseline. The type analysis, as implemented, is not particularly efficient because it passes around maps of SSA values to type sets and iterates until a fixed point is reached. This is expensive and scales poorly with program size. The analysis increases compilation time by over 100% in many cases. In the worst case, on the `earley-boyer` benchmark, the analysis incurs a compilation time slowdown of more than 100 times.

Basic block versioning does not increase compilation times by much. A limit of 5 versions per block produces a compilation time decrease of 1.2% on average, and a 5.3% increase in the worst case. It is interesting to note that in many cases, enabling basic block versioning reduces compilation time by a small amount. This is because specializing code to eliminate type checks often makes it smaller, and for some basic blocks, no machine code is generated at all.

4.7 Comparison against the V8 Baseline Compiler

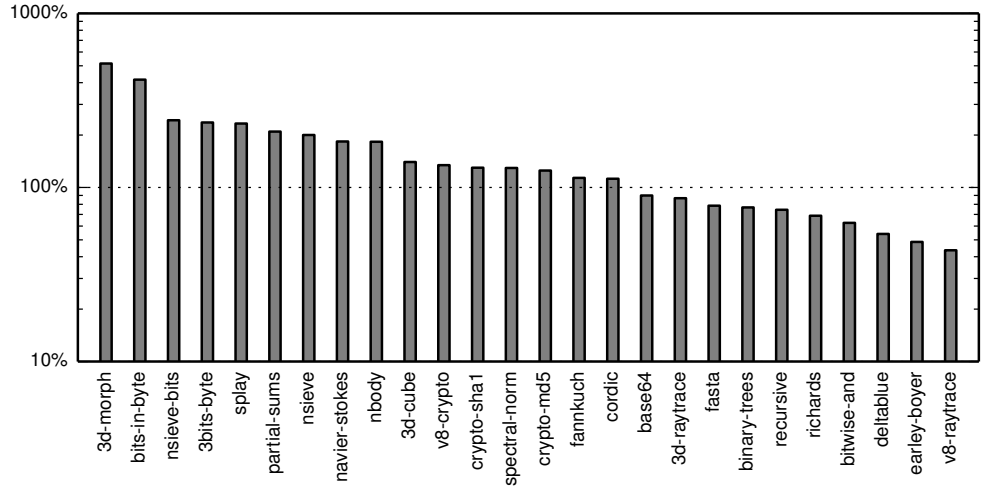

Figure 15 Speedup relative to V8 baseline (log scale, higher is better)

We have compared the execution time of the machine code generated by Higgs to that of the V8 baseline compiler. The V8 baseline compiler is not to be confused with Crankshaft. It is a low-overhead method-based JIT which, like Higgs, does not perform method inlining and only performs basic optimizations and fast on the fly register allocation. It is meant to compile code rapidly.

Figure 15 shows speedups of Higgs over V8 baseline. The scale is logarithmic, and higher bars indicate better performance on the part of Higgs. As can be seen, Higgs delivers better performance on more than half of the benchmarks. The three benchmarks on which V8 baseline does best are from the V8 suite, which the V8 baseline compiler was tailored to perform best on. Higgs is able to deliver impressive speedups on a variety of benchmarks in various areas of interest including floating point arithmetic, object-oriented data structures and string manipulation.

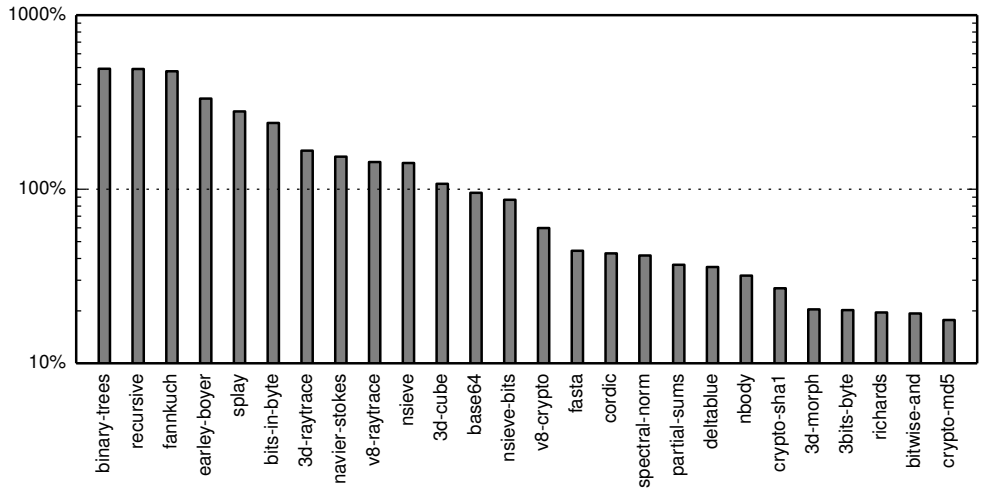

Figure 16 Speedup relative to TraceMonkey (log scale, higher is better)

4.8 Comparison against TraceMonkey

The similarity of trace compilation and basic block versioning has prompted us to compare Higgs to TraceMonkey, a tracing JIT compiler for JavaScript that was part of Mozilla's SpiderMonkey until mid 2011. It has the ability to eliminate type checks [13] based on analysis of traces. Note that Higgs does not yet implement inlining of method calls whereas TraceMonkey can inline them as part of tracing.

Figure 16 shows speedups of Higgs over TraceMonkey. The scale is again logarithmic, with higher bars indicating better performance on the part of Higgs. TraceMonkey performs better on many benchmarks. Unsurprisingly, the benchmarks TraceMonkey achieves the best performance on tend to be benchmarks which include short and predictable loops. In these, TraceMonkey is presumably able to inline all function calls which puts Higgs, without inlining, at a significant performance disadvantage.

It is interesting that Higgs, even without inlining, does much better on some of the largest benchmarks from our set. The two raytrace benchmarks, for example, make significant use of object-oriented polymorphism and feature highly unpredictable conditional branches. The `earley-boyer` benchmark is the largest of all and features complex control-flow. The `splay` and `binary-trees` benchmarks apply recursive operations to tree data structures. We note that Higgs performs much better than TraceMonkey on the `recursive` microbenchmark which suggests TraceMonkey handles recursion poorly.

Higgs shines in benchmarks with complex, unpredictable control flow as well as recursive computations. TraceMonkey is in no way the pinnacle of tracing JIT technology, but there are clearly areas where basic block versioning unambiguously wins over this implementation of trace compilation. Whereas tracing, in its simplest forms, is ideal for predictable loops, basic block versioning is not biased for any kind of control-flow structures. We believe that implementing inlining in Higgs would likely even the performance gap on the benchmarks where Higgs currently performs worse.

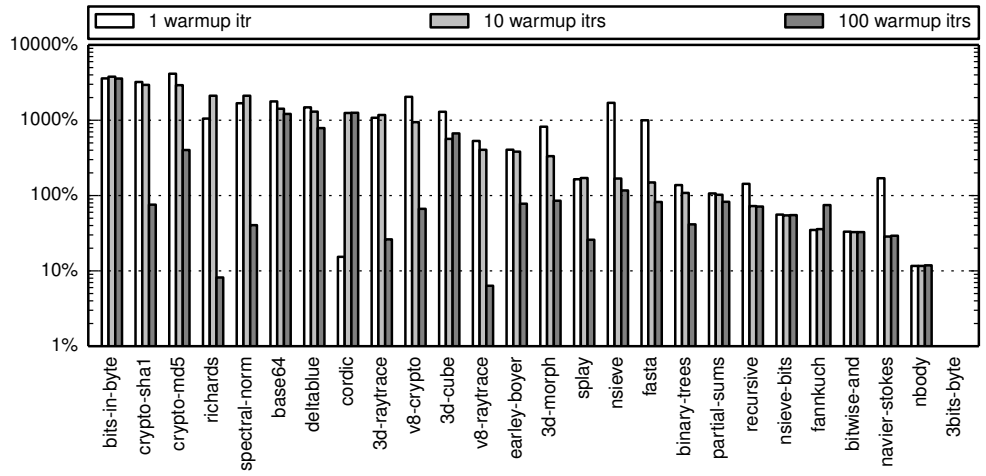

Figure 17 Speedup relative to Truffle JS (log scale, higher is better)

4.9 Comparison against Truffle JS

Figure 17 shows the relative speed of Higgs over Truffle JS on a logarithmic scale, with higher bars indicating better performance on the part of Higgs. We have evaluated the performance with 1, 10 and 100 warmup iterations. With 1 or 10 warmup iterations, Higgs outperforms Truffle on the majority of benchmarks, with speedups of up to 30x in some cases.

With 100 warmup iterations, the picture changes, and Truffle outperforms Higgs on most benchmarks. This seems to be because Truffle interprets code for a long time before compiling and optimizing it. In contrast, Higgs only needs to compile and execute a given code path once before it is optimized, with no warmup executions required.

Truffle has two main performance advantages over Higgs. The first is that after warmup, Truffle is able to perform deep inlining, as illustrated by the `v8-raytrace` benchmark. The second is that Truffle has sophisticated analyses which Higgs does not have. For instance, the recorded time for the `3bits-byte` microbenchmark is zero, suggesting that Truffle was able to entirely eliminate the computation performed as its output is never used. Doing this requires a side-effect analysis which can cope with the semantic complexities of JavaScript.

We note that even with 100 warmup iterations, and despite Truffle's powerful optimization capabilities, there remain several benchmarks where Higgs performs best, with speedups over 10x in some cases.

5 Related Work

The *tracelet-based* approach used by Facebook's HipHop VM for PHP (HHVM) [1] bears much similarity to our own. It is based on the JIT compilation of small code regions (tracelets) which are single-entry multiple-exit basic blocks. Each tracelet is type-specialized based on variable types observed at JIT compilation time. Guards are inserted at the entry of tracelets to verify at run time that the types observed are still valid for all future executions. High-level instructions in tracelets are specialized based on the guarded types. If these guards fail, new versions of tracelets are compiled based on different type assumptions and chained to the failing guards.

There are three important differences between the HHVM approach and basic block versioning. The first is that BBV does not insert dynamic guards but instead exposes and exploits the underlying type checks that are part of the definition of runtime primitives. HHVM cannot do this as it uses monolithic high-level instructions to represent PHP primitives, whereas the Higgs primitives are self-hosted and defined in an extended JavaScript dialect. The second difference is that BBV propagates known types to successors and doesn't usually need to re-check the types of local variables. A third important difference is that HHVM uses an interpreter as fallback when too many tracelet versions are generated. Higgs falls back to generic basic block versions which do not make type assumptions but are still always JIT compiled for better performance.

Trace compilation, originally introduced by the Dynamo [5] native code optimization system, and later applied to JIT compilation in HotpathVM [14] aims to record long sequences of instructions executed inside hot loops. Such linear sequences of instructions often make optimization simpler. Type information can be accumulated along traces and used to specialize code and remove type tests [13], overflow checks [28] or unnecessary allocations [7]. Basic block versioning resembles tracing in that context updating works on essentially linear code fragments and code is optimized similarly to what may be done in a tracing JIT. Code is also compiled lazily, as needed, without compiling whole functions at once.

The simplicity of basic block versioning is one of its main advantages. It does not require external infrastructure such as an interpreter to execute code or record traces. Trace compiler implementations must deal with corner cases that do not appear with basic block versioning. With trace compilation, there is the potential for trace explosion if there is a large number of control flow paths going through a loop. It is also not obvious how many times a loop should be recorded or unrolled to maximize the elimination of type checks. This problem is solved with basic block versioning since versioning is driven by type information. Trace compilers must implement parameterizable policies and mechanisms to deal with recursion, nested loops and potentially very long traces that do not fit in instruction caches.

Run time type feedback uses profiling to gather type information at execution time. This information is then used to optimize dynamic dispatch [17]. There are similarities with basic block versioning, which generates optimized code paths lazily based on types occurring at run time. The two techniques are complementary. Basic block versioning could be made more efficient by using type profiling to reorder sequences of type tests in a type dispatch. Type feedback could be augmented by using basic block versioning to generate multiple optimized code paths. The Truffle framework uses run time type feedback combined with guards to type-specialize AST nodes at run time [33, 32].

There have been multiple efforts to devise type analyses for dynamic languages. The Rapid Atomic Type Analysis (RATA) [22] is an intraprocedural flow-sensitive analysis based on abstract interpretation that aims to assign unique types to each variable inside of a function. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [4], Ruby [12] and Python [3], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [19][20][21]. Such analyses are usable in the context of static code analysis, but take too long to execute to be usable in VMs and do not deal with the complexities of dynamic code loading.

More recently, work done by Brian Hackett et al. at Mozilla resulted in an interprocedural hybrid type analysis for JavaScript suitable for use in production JIT compilers [16]. This analysis represents an important step forward for dynamic languages, but as with other type

analyses, must conservatively assign one type to each value, making it vulnerable to imprecise type information polluting analysis results. Basic block versioning can help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Basic block versioning bears some similarities to classic compiler optimizations such as *loop unrolling* [11], *loop peeling* [29], and *tail duplication*, considering it achieves some of the same results. Another parallel can be drawn with *Partial Redundancy Elimination (PRE)* [23]; the versioning approach seeks to eliminate and hoist out of loops a specific kind of redundant computation: dynamic type tests. *Code replication* has also been used to improve the effectiveness of PRE [6].

Basic block versioning is also similar to the idea of *node splitting* [30]. This technique is an analysis device designed to make control flow graphs reducible and more amenable to analysis. The *path splitting* algorithm implemented in the SUIF compiler [27] aims at improving reaching definition information by replicating control flow nodes in loops to eliminate joins. Unlike basic block versioning, these algorithms cannot gain information from type tests. The algorithms as presented are also specifically targeted at loops, while basic block versioning makes no special distinction. Mueller and Whalley have developed effective static analyses that use *code replication* to eliminate both unconditional and conditional branches [24][25]. However, their approach is intended to optimize loops and operates on a low-level intermediate representation that is not ideally suited to the elimination of type tests in a high-level dynamic language.

Customization is a technique developed to optimize Self programs [8] that compiles multiple copies of methods specialized on the receiver object type. Similarly, *type-directed cloning* [26] clones methods based on argument types, producing more specialized code using richer type information. The work of Chevalier-Boisvert et al. on *Just-in-time specialization* for MATLAB [10] and similar work done for the MaJIC MATLAB compiler [2] tries to capture argument types to dynamically compile optimized versions of whole functions. All of these techniques are forms of type-driven code duplication aimed at extracting type information. Basic block versioning operates at a lower level of granularity, allowing it to find optimization opportunities inside of method bodies by duplicating code paths.

Basic block versioning also resembles the *iterative type analysis* and *extended message splitting* techniques developed for Self by Craig Chambers and David Ungar [9]. This is a combined static analysis and transformation that compiles multiple versions of loops and duplicates control flow paths to eliminate type tests. The analysis works in an iterative fashion, transforming the control flow graph of a function while performing a type analysis. It integrates a mechanism to generate new versions of loops when needed, and a message splitting algorithm to try and minimize type information lost through control flow merges. One key disadvantage is that statically cloning code requires being conservative, generating potentially more code than necessary, as it is impossible to statically determine exactly which control flow paths will be taken at run time, and this must be overapproximated. Basic block versioning is simpler to implement and generates code lazily, requiring less compilation time and memory overhead, making it more suitable for integration into a baseline JIT compiler.

6 Limitations and Future Work

Since Higgs is a standalone JavaScript VM that is not integrated in a web browser, we have tested it on out-of-browser benchmarks that are most relevant to using JavaScript in the

server-side space (like node.js[1]). We do not anticipate any issues with using basic block versioning in a JavaScript VM integrated into a web browser, but we have not done the integration required for such an experiment. Basic block versioning is suitable for optimizing dynamic languages in general, not just JavaScript web applications in particular.

Several extensions to basic block versioning are possible. For instance, we have successfully extended it to perform overflow check elimination on loop increments, but have kept this feature disabled to simplify the presentation in this paper. Another interesting extension of basic block versioning would be to propagate information about global variable types, object identity and object property types. It may also be desirable to know the exact value of some variables and object fields, particularly for values likely to remain constant.

The implementation of lazy basic block versioning evaluated in this paper only tracks type information intraprocedurally. It would be beneficial to apply basic block versioning to function calls so that type information can propagate from caller to callee. This would entail having multiple specialized entry points for parameter types encountered at the call sites of a function. Similarly, call continuation blocks (return points) could be versioned to allow information about return value types to flow back to the caller.

7 Conclusion

We have described a simple approach to JIT compilation called lazy basic block versioning. This technique combines code generation with type propagation and code duplication to produce more optimized code through the accumulation of type information during code generation. The versioning approach is able to perform optimizations such as automatic hoisting of type tests and efficiently detangles code paths along which multiple numerical types can occur. Our experiments show that in most cases, basic block versioning eliminates significantly more dynamic type tests than is possible using a traditional flow-based type analysis. It eliminates up to 71% of type tests on average with a limit of 5 versions per block, compared to 16% for the analysis we have tested, and never performs worse than such an analysis.

We have empirically demonstrated that although our implementation of basic block versioning does increase code size in some cases, the resulting increase is quite small and pathological code size explosions are unlikely to occur. In our experiments, a limit of 5 versions per block results in a mean code size increase of just 0.19%. Our experiments with Higgs also indicate that lazy basic block versioning improves performance up to 50% with a limit of 5 versions per block. Finally, we have shown that Higgs performs better than the V8 baseline compiler on most of our benchmarks, and better than TraceMonkey on several of the more complex benchmarks in our set.

Basic block versioning is a simple and practical technique that requires little implementation effort and offers important advantages in JIT-compiled environments where type analysis is often difficult and costly. Dynamic languages, which perform a large number of dynamic type tests, stand to benefit the most.

Higgs is open source and the code used in preparing this publication is available on GitHub[2].

[1] http://nodejs.org
[2] https://github.com/higgsjs/Higgs/tree/ecoop2015

Acknowledgements

Special thanks go to Paul Khuong, Laurie Hendren, Erick Lavoie, Tommy Everett, Brett Fraley and all those who have contributed to the development of Higgs.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

References

1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 777–790. ACM New York, 2014.

2 George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI)*, pages 294–303. ACM New York, May 2002.

3 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Dynamic Languages Symposium (DLS)*, pages 53–64. ACM New York, 2007.

4 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP 2005*, pages 428–452. Springer Berlin Heidelberg, 2005.

5 V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 conference on Programming*, pages 1–12. ACM New York, 2000.

6 R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *Proceedings of the 1998 conference on Programming Language Design and Implementation (PLDI)*, pages 1–14. ACM New York, 1998.

7 Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 43–52. ACM New York, 2011.

8 Craig Chambers and David Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI)*, pages 146–160. ACM New York, June 1989.

9 Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the 1990 conference on Programming Language Design and Implementation (PLDI)*, pages 150–164. ACM New York, 1990.

10 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 46–65. Springer Berlin Heidelberg, 2010.

11 Jack W Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In *Proceedings of the 1996 international conference on Compiler Construction (CC)*, pages 59–73. Springer Berlin Heidelberg, 1996.

12 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1859–1866. ACM New York, 2009.

13 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 conference on Programming Language Design and Implementation (PLDI)*, pages 465–478. ACM New York, 2009.

14 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE)*, pages 144–153. ACM New York, 2006.

15 David Gudeman. Representing type information in dynamically typed languages, 1993.

16 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM New York, June 2012.

17 Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 conference on Programming Language Design and Implementation (PLDI)*, pages 326–336. ACM New York, 1994.

18 ECMA International. *ECMA-262: ECMAScript Language Specification*. European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edition, 2009.

19 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

20 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings 17th International Static Analysis Symposium (SAS)*. Springer Berlin Heidelberg, September 2010.

21 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 2013 Dynamic Languages Symposium (DLS)*. ACM New York, 2013.

22 Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 66–83. Springer Berlin Heidelberg, 2010.

23 E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, February 1979.

24 Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the 1992 conference on Programming Language Design and Implementation (PLDI)*, pages 322–330. ACM New York, 1992.

25 Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the 1995 conference on Programming Language Design and Implementation (PLDI)*, pages 56–66. ACM New York, 1995.

26 John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC)*, pages 566–580, 1995.

27 Massimiliano Antonio Poletto. *Path splitting: a technique for improving data flow analysis*. PhD thesis, MIT Laboratory for Computer Science, 1995.

28 Rodrigo Sol, Christophe Guillon, FernandoMagno Quintão Pereira, and Mariza A.S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In Jens Knoop, editor, *Proceedings of the 2011 international conference on Compiler Construction (CC)*, pages 2–21. Springer Berlin Heidelberg, 2011.

29 Litong Song and Krishna M Kavi. A technique for variable dependence driven loop peeling. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 390–395. IEEE, 2002.

30 Sebastian Unger and Frank Mueller. Handling irreducible loops: optimized node splitting versus dj-graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):299–333, July 2002.

31 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991.

32 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204. ACM New York, 2013.

33 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 2012 Dynamic Language Symposium (DLS)*, pages 73–82. ACM New York, 2012.

CHAPTER 6

TYPED OBJECT SHAPES

6.1 Problem and Motivation

Lazy basic block versioning, as introduced in Chapter 5, eliminates on average 71% of dynamic type checks on our benchmarks. As was shown in the paper, lazy BBV is able to outperform an intraprocedural type analysis based on a fixed point computation, both in terms of type tests eliminated and in terms of average speedups. However, the technique has two important limitations that put it at a disadvantage when compared to whole-program type analyses.

The first is that BBV, as introduced, is intraprocedural only, and cannot propagate information through function calls. The second is that it has no notion of object property types. Lazy BBV can identify which local variables are object references, but the types of properties being read from objects are always unknown. This is problematic because JS, as an object-oriented language, makes extensive use of objects. JS also defines global variables as being properties of a global object. As such, every time a global function is called, we must check that we are in fact calling a valid function, and rely on dynamic dispatch.

Ideally, we would like to have a mechanism which allows us to know the type tags associated with object properties without having to perform dynamic type tests on every property read. We would also like to be able to associate method identity information with object properties, so that when we perform method calls, we can know which method we are calling, enabling us to eliminate dynamic dispatch. Knowing the identity of methods is also very valuable for implementing interprocedural BBV, as will be explained in Chapter 7.

6.2 Whole-Program Analysis

In principle, it is possible to implement whole-program type analyses for JS. The natural thing to do, in the context of objects, is to segregate JS objects into pseudo-classes, and try to prove that specific properties of each pseudo-class must have a given type. There are a few difficulties with implementing such an analysis in JS, however. The most obvious is that of dynamic code loading and the `eval` construct, which can invalidate facts our analysis relies on. Even if we ignore this issue, and decide to only handle programs which do not employ dynamic code loading, whole-program analysis of JS code remains nontrivial.

Trying to prove facts about JS objects is difficult because the semantics of object property accesses in JS are complex and highly dynamic. There are at least two points which get in the way of static analysis. The first is that JS allows indexing objects using string keys, as if the objects were dictionaries, with the `object[key]` syntax. The unfortunate implication is that we may end up in a situation where a property of unknown type is written to an unknown object with an unknown key. This makes it difficult to segregate objects into pseudo-classes and assign types to their properties.

Another problem point when it comes to analyzing JS property accesses is that properties can be added to objects at any time, and JS allows reading missing properties from objects, which produces the special `undefined` value rather than throwing an exception. The result is that, if we cannot prove that a given property must be defined on an object when reading it, then we do not know that the value read will not be `undefined`. This results in situations where `undefined` flows into various calculations and pollutes analysis results.

We have attempted to implement a static whole-program type analysis for JS inspired by [25, 26] as part of the Tachyon project (see Chapter 3), with the goal of eventually using this analysis to optimize programs. The results were unfortunately somewhat disappointing, with the analysis taking up to several minutes to execute on some programs. We also had limited success in terms of analysis precision. It is very difficult, even with path-sensitivity and context information, to prove ahead of time that object prop-

erties must be defined on every path from an object's initialization to the point where a property is read.

6.3 Typed Shapes

All modern JS engines have some concept of maps or shapes (see Section 2.4). That is, objects have an associated piece of metadata which tells us which properties the objects define, the memory offset at which the property is located, as well as which special attributes the property has (e.g. read-only, non-enumerable). As such, it is fairly natural to think that shapes could be extended to also encode property types. Such an extension was implemented by the Truffle Object Storage Model (OSM) [37], which can encode property type tags as part of object shapes.

Encoding property type information in object shapes is convenient because JS engines already rely on dynamic dispatch (inline caches) at property access sites to establish a correspondence between object shapes and the memory offsets at which properties are stored. As such, once we have estalished the shape of an object at a property read, we also get type information about the property being read, at no extra cost. The tradeoff is that we must check the types of properties being written. This cost is easily amortized, however, because typical JS programs have much more property reads than writes.

The paper presented in this chapter describes how we implement a system of typed object shapes inspired from the Truffle OSM. Our system encodes type tags, and also method identity information. We incorporate typed shapes with BBV by extending it to propagate object shape information as part of the normal versioning scheme. This allows us to effectively eliminate dynamic type tag tests after property reads, but also to eliminate redundant dynamic shape tests. Furthermore, because BBV is very effective at propagating value types, we are able to eliminate most type tests at property writes.

6.4 Results

The results obtained are very encouraging. By making BBV aware of object property types, we have been able to eliminate 48% more type tests than was possible using

intraprocedural BBV alone, and achieve further speedups. In addition to this, the encoding of method identity information in typed shapes allows BBV to know the identity of callees for 90% of calls.

This paper was submitted to CGO 2016, but was unfortunately not accepted. We have chosen to include it in this thesis because it provides a detailed explanation of the workings of typed object shapes, which are central to our implementation of interprocedural BBV (see Chapter 7). The article is publicly available through the arxiv.org website [8].

Extending Basic Block Versioning with Typed Object Shapes

Maxime Chevalier-Boisvert
DIRO, Université de Montréal, Quebec, Canada

Marc Feeley
DIRO, Université de Montréal, Quebec, Canada

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization, code generation, run-time environments

Keywords Just-In-Time Compilation, Dynamic Language, Optimization, Object Oriented, JavaScript

Abstract

Typical JavaScript programs feature a large number of object property accesses. Hence, fast property reads and writes are crucial for performance. Unfortunately, many (often redundant) dynamic checks are implied in each property access and the semantic complexity of JS makes it difficult to optimize away these tests through program analysis.

We introduce two techniques to effectively eliminate a large proportion of dynamic checks related to object property accesses. *Typed shapes* enable code specialization based on object property types without potentially complex and expensive analyses. *Shape propagation* allows the elimination of redundant shape checks in inline caches. These two techniques combine particularly well with basic block versioning, but should be easily adaptable to tracing JIT compilers and method JITs with type feedback.

To assess the effectiveness of the techniques presented, these were implemented in Higgs, a type-specializing JIT compiler for JS. Across the 26 benchmarks tested, the techniques eliminate on average 48% more type tests, reduce code size by 17% and reduce execution time by 25% compared to a baseline using untyped shapes. On several benchmarks, Higgs extended with the techniques presented outperforms Truffle/JS.

1. Introduction

Typical JavaScript (JS) programs make heavy use of object property accesses. The highly complex and highly dynamic semantics of JavaScript mean that each property access contains several implicit dynamic checks (Section 2.1). Generating efficient machine code means eliminating as many of these checks as possible.

The Truffle Object Storage Model (OSM) [24] introduces the idea of specializing object shapes by using them to encode property types. This proves useful in eliminating some dynamic type checks after property reads, but many more dynamic checks still remain.

Basic Block Versioning (BBV) [7] is a JIT compilation strategy which allows rapid and effective generation of type-specialized machine code without a separate type analysis pass.

The first contribution of this paper (Section 3.1) is the design of a typed object shape model similar to the Truffle OSM as a natural extension of BBV, allowing BBV to introspect object types.

The second contribution of this paper (Section 3.3) is to leverage the strengths of BBV and use it to propagate object shape information, making it possible to optimize Polymorphic Inline Caches (PICs) by eliminating redundant dynamic shape tests.

The third contribution of this paper (Section 3.2) is the extension of the typed shape model to encode not only type tags, but also method identity information, which is particularly useful to optimize JS code.

Typed shapes and shape propagation were implemented in Higgs[1], a JIT compiler for JS built around BBV. Empirical results across 26 benchmarks (Section 4) show that, on average, the techniques introduced eliminate 48% more type tests, reduce code size by 17% and reduce execution time by 25%. The performance of Higgs with typed shapes and shape propagation is competitive with that of Truffle/JS, outperforming it on several benchmarks.

[1] https://github.com/higgsjs/Higgs

2. Background

2.1 JavaScript Objects

JavaScript objects follow a prototype-based inheritance model [14] inspired from Self [6]. Objects can dynamically grow, meaning that properties can be added to or deleted from objects at any time. The types of properties are not constrained, and properties can be redefined to have any type at any time. Semantically, JS objects can be thought of as behaving somewhat like hash tables, but the semantics of property accesses are complex.

In JS, object properties can be plain values or accessor (getter/setter) methods which may produce side-effects when executed. Individual object properties can have read-only (constant) attribute flags set, which prevents their redefinition. When a property is not defined on an object, the lookup must traverse the prototype chain recursively. These factors mean that each JS property read or write implies multiple hidden dynamic tests. Ideally, most of these tests should be optimized away to maximize performance.

Global variables in JS are stored on a first-class global object, which behaves like any other. Properties of the global object can thus also be defined to be read-only, or be accessor methods. Hence, optimizing global property accesses is also nontrivial. Since the global object is a singleton and typically large in size, modern JS engines such as Google's V8 tend to implement it using a different strategy from regular objects.

2.2 Object Shapes

JS objects can be thought of as behaving like hash maps associating property name strings to property values and attribute flags. However, implementing objects using hash maps is inefficient both in terms of memory usage and property access time. Doing so means that each object must store a name string, a value and attribute flags for each property. Furthermore, each property access must execute a costly hash table lookup which may involve repeated indirections.

High-performance JS engines (V8, SpiderMonkey, etc.) rely on the concept of object shapes, also known as "hidden classes". This approach aims to exploit the fact that programs typically create many objects with the same properties, that objects are usually initialized early in their lifetime, and that property deletions and additions after initialization are infrequent.

Shapes are object layout descriptors. They are composed of shape nodes, with each shape node containing the name, memory offset and attribute flags for one property. All existing shape nodes are part of a global tree structure representing the order in which properties have been added to objects. Each object contains a shape pointer which points to a shape node representing the last property added to the said object. All objects are initially created with no properties and begin their lifetime with the empty shape. Adding a property to an object updates its shape pointer.

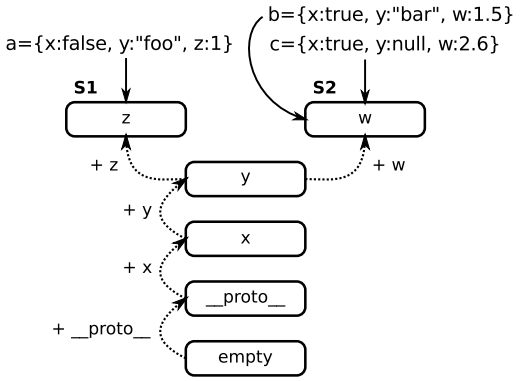

Figure 1. Object shapes as part of a shape tree

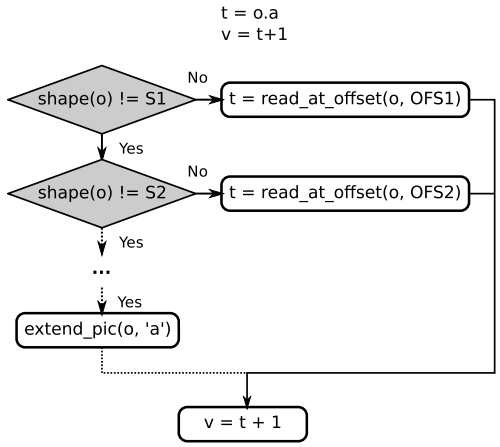

Figure 2. Property read using a Polymorphic Inline Cache (PIC)

Figure 1 illustrates the shape nodes for three different JS object literals. All objects have a hidden `__proto__` property which stores a pointer to the prototype object. All three objects also share properties named `x` and `y`, hence, part of the shapes of these two objects are made of the same shape nodes. The last property added to object `a` is `z`, and so it has shape `S1`. The last property added to objects `b` and `c` is `w`, and so these have shape `S2`.

2.3 Polymorphic Inline Caches

Object shapes are more space efficient than using hash maps, since multiple objects with the same set of properties and initialization order can share the same shape. However, shapes, by themselves, do not make property accesses faster. Naively traversing the shape structure of an object on every property access is highly inefficient.

Polymorphic Inline Caches (PICs), pioneered in the implementation of the Self programming language [6, 13], are commonly used to accelerate property accesses. They are used by modern JS VMs such as V8 and SpiderMonkey. The core idea is to generate machine code on the fly to determine

the shape of an object and generate an efficient dispatch at each property access site. This machine code takes the form of a cascade of shape test operations and is updated as new object shapes are encountered. PICs can be thought of as offloading the property lookup overhead to code generation time instead of execution time.

Figure 2 illustrates a property read implemented using a PIC. Two shape tests match previously encountered object shapes. Each test, if it encounters a matching shape, triggers the execution of a load machine instruction which reads the property from the object at the correct memory offset. This memory offset is determined at code generation time based on the object's shape, which tells us where each property is located. When objects passing through a property read have only one possible shape, a property read can be as fast as one comparison and one load instruction.

2.4 Basic Block Versioning

Basic Block Versioning (BBV) is a JIT code generation technique originally applied to JavaScript by Chevalier-Boisvert & Feeley [7], and adapted to Scheme by Saleil & Feeley [21]. The technique bears similarities to HHVM's tracelet-based compilation approach and Psyco's JIT code specialization system [20].

Efficient type-specialized machine code is generated in a single pass, without the use of costly type inference analyses or profiling. BBV achieves this by lazily cloning and type-specializing single-entry single-exit basic blocks on the fly. As in Psyco, code generation and code execution are interleaved so that run-time type information can be extracted by the code generation engine. The accumulated information allows the removal of redundant type tests, particularly in performance-critical paths.

Higgs segregates values into a few categories based on type tags [11]. These categories are: 32-bit integers (`int32`)[2], 64-bit floating point values (`float64`), miscellaneous JS constants (`const`), and four kinds of garbage-collected pointers inside the heap (`string`, `object`, `array`, `closure`). These type tags form a simple, first-degree notion of types that is used to drive code versioning.

BBV, as introduced in [7], deals only with function parameter and local variable types. It has no mechanism for handling object property types and global variable types. The current work extends BBV to include a more advanced notion of object types based on *typed shapes*, and enable type-specialization based on object property and global variable types.

3. Typed Shapes

The main contributions of this paper are presented here.

[2] Note that while according to the ES5 specification all JavaScript numbers are IEEE double-precision floating point values, high-performance JavaScript VMs typically attempt to represent small integer values using machine integers so as to improve performance by using lower latency integer arithmetic instructions.

```
var enable_debug = function F3() {
  debug = function F2(msg) { alert(msg); };
}
var debug = function F1() {};
debug("start"); // callee is known to be F1
enable_debug();
debug("again"); // callee is known to be F2
```

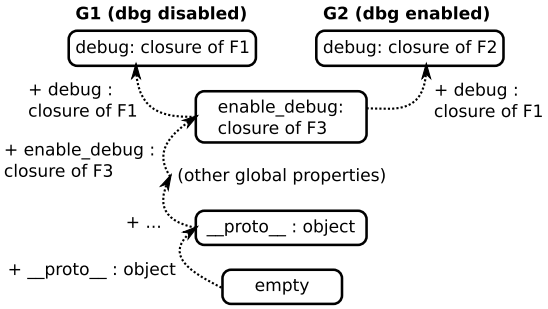

Figure 4. Object shapes for global properties and functions

3.1 Typed Shapes and Property Types

Object shapes in other JS engines encode property names, slot indices and meta-information such as attribute flags (writable, enumerable, etc.). We extend shapes to also encode property types: this makes it possible for us to specialize code based on the types of property values. Testing the shape of an object once gives us the type of all its properties.

Figure 3 shows the object shapes associated with three different JS object literals. Shape nodes are now annotated with type tags corresponding to property values. Objects `b` and `c` share the same property names, but the type of their `y` property differs. The property `b.y` is a string, whereas `c.y` has value `null` which has type tag `const`.

Our definition of typed shapes is not recursive. Shapes corresponding to property values which are object references do not encode the shape of the object being referenced. This is because objects are mutable. Hence, if `a.b` is an object, its shape cannot be guaranteed to remain the same during the execution of a program, but objects will always remain objects, so the type tag of `a.b` will not change so long as this property is not overwritten.

With typed shapes, type tags are encoded in the shape and so do not need to be encoded in objects themselves, making it possible for property values to be stored in an unboxed representation, avoiding boxing and unboxing overhead. In the optimal case, properties can be read and written in a single machine instruction.

A further advantage is that the shape of an object can tell us whether or not the object has a prototype or not. This eliminates the need to perform a `null` check when going up the prototype chain during a property read.

3.2 Method Identity and the Global Object

The property type information currently encoded in typed shapes includes type tags, but also function pointers (function/method identity). Encoding function pointers makes it

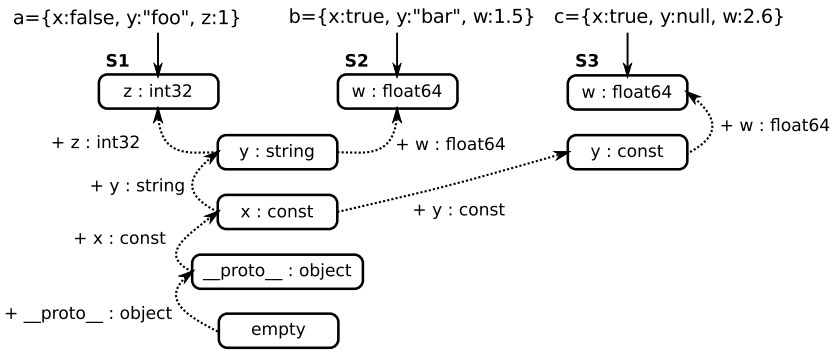

Figure 3. Type meta-information on object shapes and property additions

possible to know the identity of callees at call sites. This information is highly valuable for optimization purposes. For instance, when the identity of a callee is known at a call site, passing unused argument values (such as the hidden `this` argument) can be avoided.

Our approach uses a unified implementation for all objects, including the global object. This makes it possible to optimize global property accesses using the same techniques as regular property accesses. This contrasts with V8, which uses a collection of individual mutable cells to implement its global object.

Figure 4 illustrates the global object shape in relation to a snippet of code where a call to `enable_debug` replaces an inactive implementation of the `debug` function by one which displays error messages. The `enable_debug` function causes the global object to switch from shape `G1` to `G2`. The global object shape encodes the identity of functions, meaning that for both calls to `debug`, the identity of the function is known at code generation time, avoiding the need for dynamic dispatch.

3.3 Shape Propagation

As described in Section 2.3, Polymorphic Inline Caches (PICs) are a lazily generated chain of dynamic tests to identify an object's shape and quickly select a fast implementation of a property read or write. Typical PIC implementations treat them as monolithic blocks of machine code which are an internal part of property access instructions in the compiler's Intermediate Representation (IR).

To integrate typed shapes with BBV, PICs are divided into their component parts, exposing them as explicit chains of individual shape tests in our IR. This allows BBV to extract shape information from shape tests and propagate known object shapes from one property access site to another by leveraging existing BBV type propagation mechanisms. Propagating shapes allows eliminating redundant (repeated) shape tests in successive property accesses on the same object.

As shown in [7], most Static Single Assignment (SSA) values are monomorphic in terms of type tags. Few values are polymorphic. This remains true when shapes come into the picture. Most program points see only one shape for a given value. However, the objects which are polymorphic in shape are sometimes megamorphic. That is, one property access site can receive objects of a large number of different shapes. This can lead to situations where rare megamorphic values cause disproportional code size growth. To avoid this problem, shape propagation is limited to tracking one possible shape per SSA value.

3.4 Guarding Property Writes

Overwriting a property value may cause an object to transition to a new shape if the type of the new value doesn't match the type encoded in the object's current shape. For instance, if object `c` from Figure 3 was to have its `y` property overwritten with a string value (e.g. `c.y = "bif"`), then the shape of `c`, which was previously `S3`, would change to `S2`.

As such, in a system with typed shapes, property writes have implicit guards on the type of values written. Fortunately, it's possible to eliminate most such guards because values types are often known when compiling property writes (see Section 4.1). It's also the case that shape changes are relatively rare, and incur little overhead. Assuming that the new shape has been previously allocated and initialized, which is often the case, changing an object's shape is only a matter of overwriting the object's shape pointer, which can be done in a single machine instruction.

3.5 A Simple Example

With polymorphic inline caches, reading or writing to an object property implies first performing a number of dynamic checks to dispatch read or write operations based on the object shape (see section 2.3). Many JS primitives, including arithmetic operators, also perform dynamic dispatch based on value types.

Figure 5 illustrates the operations involved in incrementing the value of an integer property on an object (`a.z = a.z + 1`) when using traditional PICs (without typed shapes). There are four dynamic checks. A first check is performed to dispatch based on the object shape when reading the property. A second check is performed to dispatch based on the

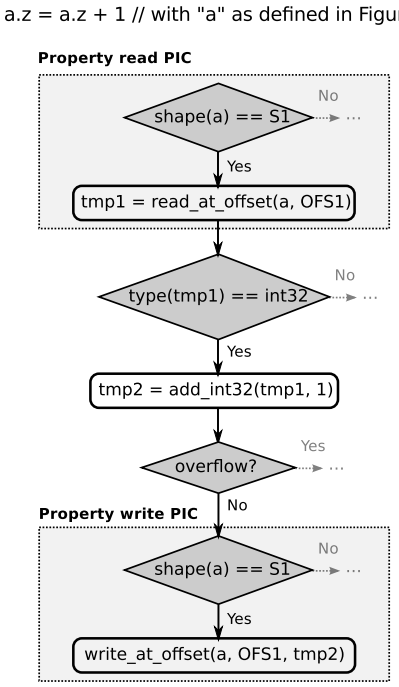

Figure 5. Operations involved in a property read and write with traditional Polymorphic Inline Caches (PICs)

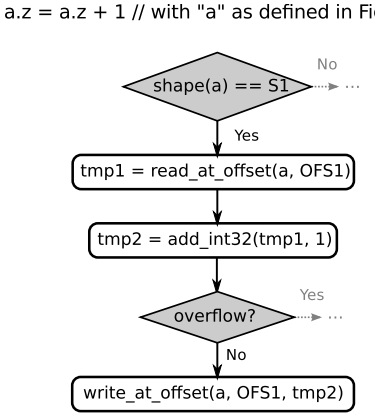

Figure 6. Operations involved in a property read and write with typed shapes and shape propagation, starting with an object of unknown shape

type of the property's value (which is `int32` in this case). A third check is performed to verify that the result of the integer addition operation did not result in an integer overflow. Finally, a fourth dynamic check is performed when writing back the incremented value. This last check is necessary because the property read and property write PICs are distinct.

Typed shapes and shape propagation produce more efficient code, as illustrated in Figure 6. The dynamic dispatch based on the property type is eliminated, because this type is encoded in the object's shape, and is thus automatically known once the object's shape has been tested. The dispatch based on the object's shape when writing back the property is eliminated because the object's shape was previously tested and this information is propagated to the write. There is no need to guard the type of `tmp2` when writing the new property value because this type is deduced based on the type of `tmp1`.

4. Evaluation

An implementation of the Higgs JIT compiler extended with typed shapes and shape propagation was tested on a total of 26 classic benchmarks from the SunSpider and V8 suites. One benchmark from the SunSpider suite and one from the V8 suite were not included in our tests because Higgs does not yet implement the required features. Benchmarks making use of regular expressions were discarded because Higgs and Truffle/JS [25, 26] do not implement JIT compilation of regular expressions.

To measure steady state execution time separately from compilation time in a manner compatible with both Higgs and Truffle/JS, the benchmarks were modified so that they could be run in a loop. A number of warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place.

The number of warmup and timing iterations were scaled so that short-running benchmarks would execute for at least 1000ms in total during both warmup and timing. Unless otherwise specified, all benchmarks were run for at least 10 warmup iterations and 10 timing iterations.

Truffle/JS v0.5 was used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 CPU and 16GB of RAM running Ubuntu Linux 12.04. Dynamic CPU frequency scaling was disabled to ensure reliable timing measurements.

4.1 Type Tag Tests

Encoding type tags in object shapes implies that property writes must be guarded with type tag checks. Fortunately, with BBV, the type tag of values is most often known at code generation and does not need to be dynamically tested. As a result, just 7% of property writes need to be guarded on average across benchmarks. While guards increase the number of dynamic type tests slightly, typed shapes make it possible to eliminate type tag tests at property reads, and there are 11.7 property reads for every property write on average.

Figure 7 shows the total number of type tag tests (including guards on property writes) performed with typed shapes and with typed shapes coupled with shape propagation relative to a baseline which uses traditional inline caches without typed shapes or shape propagation. The chart makes it clear that typed shapes reduce the number of type tests executed very significantly, by 47% on average. In the case of

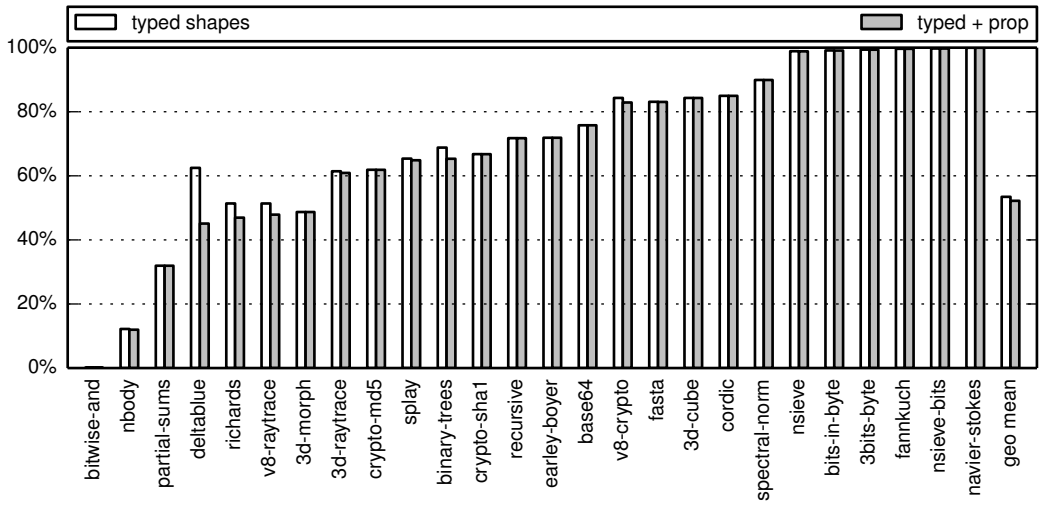

Figure 7. Number of type tests relative to inline cache baseline

the `bitwise-and` microbenchmark, which operates entirely on global variables, type tests are reduced by nearly 100%.

Note that enabling shape propagation produces a slight reduction in type tests on some benchmarks. This is because enabling the propagation of shapes allows eliminating a `null` check while traversing the prototype chain, since the prototype link is itself represented as a typed property. The benchmarks which benefit the most from this phenomenon are those which make heavy use of inheritance.

4.2 Shape Tests

Figure 8 illustrates the number of shape tests relative to a baseline using polymorphic inline caches (without typed shapes or shape propagation). Notably here, enabling typed shapes increases the total number of dynamic shape tests by 17% on average. This is because enabling the specialization of shapes based on property types necessarily result in more shape polymorphism at run time. Hence, individual inline caches tend to see longer chains of shape tests.

Enabling shape propagation produces a reduction in the number of shape tests when compared to baseline, by 19% on average. Hence, shape propagation effectively mitigates the increase in shape tests resulting from typed shapes, This is because there are many instances where multiple property reads on the same object occur within a given function, and shape propagation can allow eliminating further shape tests after the first property access on an object.

4.3 Function Calls

Without typed shapes, Higgs does not know the identity of callees at most call sites. With typed shapes, on average, callee identity is known for 90% of calls executed. For most benchmarks, the callee identity is known for all calls. There are some exceptions because at present Higgs cannot introspect captured closure variables or closures passed as function arguments.

4.4 Machine Code Size

Typed shapes allow us to eliminate type checks and generate more efficient code, which tends to be more compact. As a result, with typed shapes, the size of generated machine code is smaller on every benchmark when compared to a baseline without typed shapes, with an average code size reduction of 16%. Enabling shape propagation allows us to eliminate redundant shape tests, and yields a 17% average code size reduction over baseline.

4.5 Compilation time

Typed shapes and shape propagation do not affect compilation time significantly in our system. In the worst case, with typed shapes and shape propagation, compilation time increases by 5%, but the average change is on the order of 1%.

4.6 Execution Time

Figure 9 shows the execution times obtained with typed shapes and shape propagation relative to a baseline using inline caches only (untyped shapes). Typed shapes alone produces an average execution time reduction of 21% over untyped shapes. Enabling shape propagation yields an average execution time reduction of 25%, a slight but significant improvement over typed shapes alone.

The `bitwise-and` microbenchmark, using only global variable accesses in a small loop with no shape polymorphism, is an ideal showcase for typed shapes and shape propagation. However, shape propagation makes little performance difference on some benchmarks. This is in part because the current implementation cannot preserve known

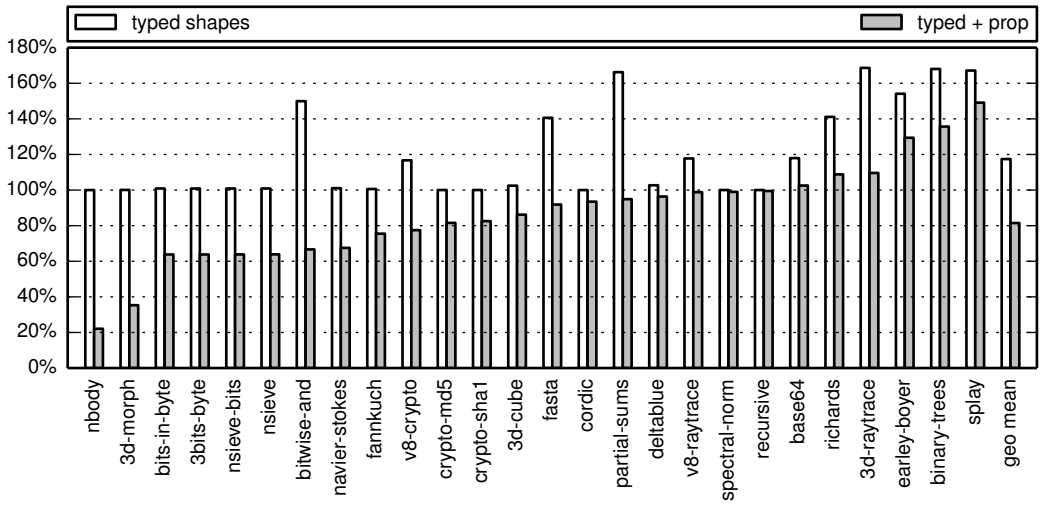

Figure 8. Number of shape tests relative to inline cache baseline

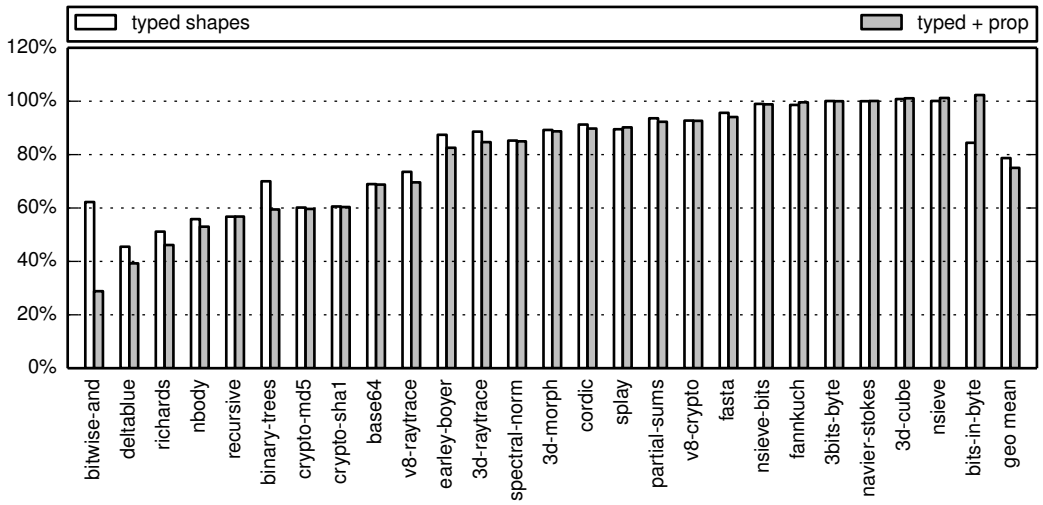

Figure 9. Execution time relative to inline cache baseline (lower is better)

shapes across function calls. Overcoming this limitation is part of future work (see Section 5).

4.7 Comparison with Truffle/JS

This paper would not be complete without a comparison of the performance of Higgs against another JavaScript implementation. The performance of Truffle/JS was compared against that of Higgs. Truffle/JS is an ideal comparison point as it is, like Higgs, a research VM written in a garbage-collected language. Truffle/JS also implements a typed shape system very similar to that presented here.

Since Truffle takes a relatively long time to compile and optimize code, each benchmark was run for 1000 "warm up" iterations before measuring execution time, so as to more accurately measure the final performance of the generated machine code once a steady state is reached. After the warm up period, each benchmark is run for 100 timing iterations.

Figure 10 shows the results of our performance comparison against Truffle/JS. The left column represents the speedup of Higgs over Truffle when taking only execution time into account, with the warmup iterations excluded. The right column is the speedup of Higgs over Truffle/JS when comparing total time, including initialization, compilation and warmup. A logarithmic scale was used due to the wide spread of data points across multiple orders of magnitude.

On average, when taking only execution time into account, Higgs outperforms Truffle on 13 out of 26 benchmarks and is 28% faster than Truffle/JS on average. When

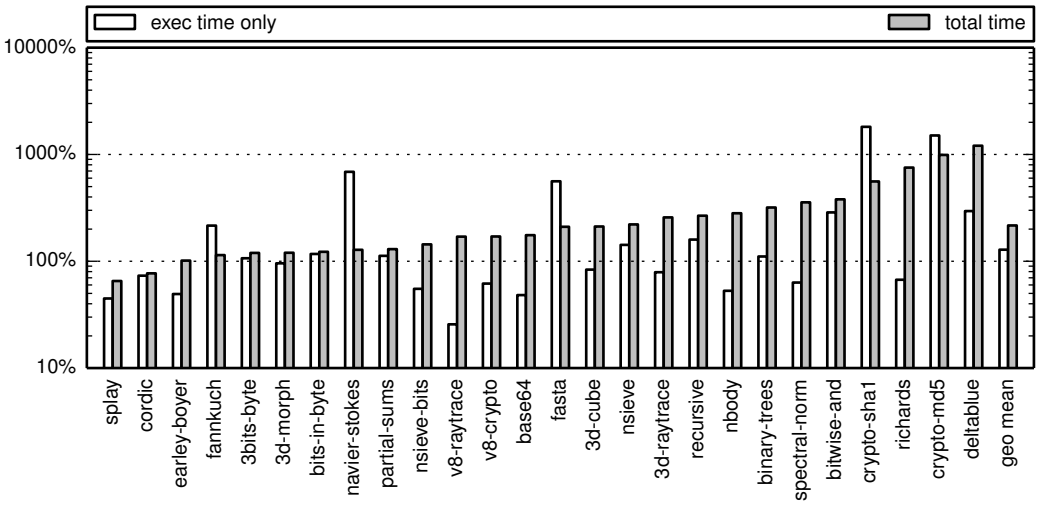

Figure 10. Speed relative to Truffle/JS (log scale, bars above 100% favor Higgs)

comparing wall clock times, Higgs outperforms Truffle on the majority of benchmarks, and is 116% faster on average.

It is not the case that Higgs outperforms Truffle/JS on every benchmark. JIT compiler design is a complex problem space, and as a result, there are benchmarks where each system outperforms the other by a wide margin. Truffle/JS is a method-based compiler implementing profiling, type-feedback and type analysis. It has a few technical advantages over Higgs, such as the use of method inlining, a more sophisticated register allocator, multithreaded background compilation, and a highly optimized garbage collector implementation (the Java VM's). These tools give Truffle an edge on specific benchmarks, such as `v8-raytrace`. Nevertheless, Higgs produces competitive machine code in many cases.

It is interesting to note that Higgs performs much better than Truffle/JS on the `bitwise-and` microbenchmark, indicating that Higgs likely has faster global variable accesses than Truffle/JS Higgs also outperforms Truffle/JS on the `recursive` microbenchmark, which takes advantage of known callee identities provided by typed shapes.

5. Future Work

Work by Costa, Alves et al. [22] has shown that significant speedups can be obtained by specializing JS code based on function argument values, which are often constant. Similarly, typed shapes could be extended to allow for the direct encoding of constants into object shapes. This would likely be useful for global variables which are never mutated and effectively constant.

An important limitation of the shape propagation approach as presented in this paper is that it is intraprocedural only. Known shapes are not propagated to callees, and furthermore, known shape information is lost whenever a function call is made. This is because function calls are currently treated as black boxes, and it is not guaranteed that callees will not change the shape of objects used in the caller.

It may be interesting to investigate interprocedural basic block versioning. Specifically, it may be useful to specialize function entry points so that known types can be propagated from callers to callees. This would contribute to eliminating more type tests and shape tests. Typed shapes will make the implementation of interprocedural BBV easier and more efficient, since they provide precise information about callee identities.

Having information about callee identities should also make it possible to implement a rudimentary analysis to assess whether or not callees modify object shapes or not. Having a way to guarantee that a callee will not cause any object to change shape makes it possible to avoid discarding shape information at call sites, thereby improving the effectiveness of shape propagation.

6. Related Work

Polymorphic Inline Caches (PICs) were originally introduced in literature discussing the efficient implementation of the Self programming language [6, 13]. Self did not use shapes exactly as discussed in this paper, but instead a concept of *maps* which grouped objects cloned from the same prototype. These served the same purpose as shapes, reducing memory usage overhead and storing metadata relating to properties (though not type information). Today, Commercial JS implementations such as Google's V8, Mozilla's SpiderMonkey, Apple's Nitro and Oracle's Truffle/JS [25, 26] all implement something equivalent to PICs and object shapes to accellerate property accesses.

The Truffle Object Storage Model (OSM) [24] describes a typical implementation of an object system where each ob-

ject contains a pointer to its shape, which describes the layout of the object (property locations) and property attribute metadata. The OSM introduces the notion of specializing shapes based on types to the literature. We show how to effectively integrate such a model with a compiler based on BBV, and extend upon it with the notion of shape propagation, taking advantage of the capabilities of BBV.

Several *whole-program type analyses* for JS were developed [15–17]. These analyses are generally considered too expensive to use in a JIT compiler. They also tend to suffer from precision limitations when dealing with object types. It is often difficult, for instance, to prove that a specific property of an object must be initialized at a given program point. Work done by Kedlaya, Roesch et al. [18] shows strategies for improving the precision of type analyses by combining them with type feedback and profiling. The strategy shows promise, but does not explicitly deal with object and property types.

Work by Hackett and Guo at Mozilla resulted in an interprocedural hybrid type analysis for JS suitable for use in production JIT compilers [12]. This approach has a notion of object types segregating objects by prototype, and tries to bound types a given property associated with a given object type may have. The Mozilla approach does not always guarantee that a given property has a given type, and so often cannot unbox property values. It also relies on a supplemental analysis which examines constructor function bodies to try and prove property initialization. The approach presented here is simpler and potentially more precise.

Trace compilation, originally introduced by the Dynamo [2] native code optimization system, and later applied to JIT compilation in HotpathVM [9] aims to record long sequences of instructions executed inside hot loops. Such linear sequences of instructions often make optimization simpler. Type information can be accumulated along traces and used to specialize code and remove type tests [8], overflow checks [23] or unnecessary allocations [3].

The *TraceMonkey* tracing JIT compiler for JS can specialize traces based on types [8]. It can also guard based on object shapes and eliminate some shape dispatch overhead inside traces, similarly to the shape propagation discussed in this paper. It does not, however specialize code based on property types. Trace compilation [4] and meta-tracing are an active area of research [5] in the realm of dynamic language optimization. Most tracing JIT compilers for languages which have some concept of objects, tuples or records could likely benefit from the approaches discussed in this paper.

Facebook's *HHVM* for PHP [1] uses an approach called Tracelet specialization which has many similarities with BBV. Since PHP is an object-oriented dynamic language and HHVM already specializes code using type guards, it seems this system could likely benefit from typed shapes and shape propagation.

Grimmer, Matthias et al. [10] implemented an interpreter which can access C structs and arrays as JS objects at better speeds than native JS objects. This is useful when interfacing with C, but likely impractical as a drop-in replacement for JS objects. There is a proposal for the inclusion of *typed objects* (also known as "struct types") in ECMAScript 7, a future revision of the JS language. These are objects using pre-declared memory layouts with type-annotated fields, much like C structs. One of the stated goals is to improve optimization opportunities for JIT compilers [19].

7. Conclusion

Two techniques to effectively specialize code based on object and property types were described. Typed shapes, an extension to the familiar object shapes used in most commercial JS engines, enables the elimination of type tag checks after property reads. Shape propagation allows the elimination of redundant shape checks, reducing the overhead of Polymorphic Inline Caches (PICs).

Across the 26 benchmarks tested, when compared to a baseline without typed shapes, the combination of these two techniques eliminate on average 48% more type tests and 19% of shape tests. Code size is reduced by 17% and execution time by 25%. An implementation of Higgs extended with typed shapes and shape propagation performs competitively with Truffle/JS, outperforming it on several benchmarks.

The techniques presented combine particularly well with a compiler architecture based on BBV, but should be easily adaptable to compilers based on trace compilation or method JITs with type feedback.

An artifact including complete source code will be submitted to the CGO artifact evaluation committee so that our results can be replicated.

Acknowledgements

Special thanks go to Laurie Hendren, Erick Lavoie, Vincent Foley, Paul Khuong, Molly Everett, Brett Fraley and all those who have contributed to the development of Higgs.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

References

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 777–790. ACM New York, 2014.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 conference on Programming*, pages 1–12. ACM New York, 2000.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 43–52. ACM New York, 2011.

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin 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*, pages 18–25. ACM, 2009.

[5] Carl Friedrich Bolz, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. Meta-tracing makes a fast Racket. *Workshop on Dynamic Languages and Applications*, 2014.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, September 1989.

[7] Maxime Chevalier-Boisvert and Marc Feeley. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101–123. Schloss Dagstuhl, 2015. `http://arxiv.org/abs/1411.0352`.

[8] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[9] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE)*, pages 144–153. ACM New York, 2006.

[10] Matthias Grimmer, Thomas Würthinger, Andreas Wöß, and Hanspeter Mössenböck. An efficient approach for accessing C data structures from javascript. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, pages 1:1–1:4, New York, NY, USA, 2014. ACM.

[11] David Gudeman. Representing type information in dynamically typed languages, 1993.

[12] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM New York, June 2012.

[13] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[14] ECMA International. *ECMA-262: ECMAScript Language Specification*. European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edition, 2009.

[15] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings 17th International Static Analysis Symposium (SAS)*. Springer Berlin Heidelberg, September 2010.

[17] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 2013 Dynamic Languages Symposium (DLS*. ACM New York, 2013.

[18] Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. *SIGPLAN Not.*, 49(2):37–48, October 2013.

[19] Nicholas D. Matsakis, David Herman, and Dmitry Lomov. Typed objects in javascript. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 125–134, New York, NY, USA, 2014. ACM.

[20] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM.

[21] Baptiste Saieil and Marc Feeley. Code versioning and extremely lazy compilation of scheme. In *Scheme and Functional Programming Workshop*, 2014.

[22] Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[23] Rodrigo Sol, Christophe Guillon, FernandoMagno Quinto Pereira, and Mariza A.S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In Jens Knoop, editor, *Proceedings of the 2011 international conference on Compiler Construction (CC)*, pages 2–21. Springer Berlin Heidelberg, 2011.

[24] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 133–144, New York, NY, USA, 2014. ACM.

[25] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming*

& Software, Onward! 2013, pages 187–204. ACM New York, 2013.

[26] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 2012 Dynamic Language Symposium (DLS)*, pages 73–82. ACM New York, 2012.

CHAPTER 7

INTERPROCEDURAL BASIC BLOCK VERSIONING

Typed shapes, as presented in Chapter 6, enable BBV to encode information about object property and global variable types. With this addition, lazy BBV is able to eliminate 79% of dynamic type tests. The technique is still limited in one important aspect, which is that it treats function calls like black boxes. The paper presented in this chapter explains the mechanisms we have devised to address this limitation. This paper has been submitted to ECOOP 2016 and is still under review. In the meantime, it is publicly available through arxiv.org [6].

7.1 Problem and Motivation

BBV, as we have presented it so far, assumes that all function parameters have unknown types when entering a function. This seems wasteful, because in most cases, the types of function arguments are probably known at call sites. The result is that we are performing many dynamic type tests which could potentially be avoided. The same could be said about function return values, which are always assumed to be of unknown types in callers.

The recursive Fibonacci function (see Figure 7.1) exemplifies this. Here, it is obvious to a programmer that given an integer input `n`, the `fib` function is only ever called on integer values. As such, no dynamic type tests should be needed. However, with intraprocedural versioning as presented thus far, the type of the parameter `n` is tested on each call. The type of values returned by `fib` is also tested twice per call. To compute `fib(40)`, over 662 million `is_int32` type tag tests are performed.

7.2 Interprocedural Versioning

A straightforward extension of BBV is to pass type information from callers to callees simply by allowing for multiple specialized versions of function entry points.

```
function fib(n)
{
    if (n < 2)
        return n;

    return fib(n-1) + fib(n-2);
}

fib(40);
```

Figure 7.1: The recursive fibonacci function

This allows callers to propagate the argument types they known to callees, which receive the appropriate parameter type information. By using callee identity information provided by typed shapes (Chapter 6), entry point versioning can be done without dynamic dispatch. That is, if the identity of a callee is known, it becomes possible for the caller to jump directly to a specialized entry point in the callee which encodes information about parameter types, with zero dynamic overhead.

Similarly, passing type information from callees to callers can be achieved by specializing call continuations in callers. That is, we can create specialized versions of the basic blocks to which the callee function returns which have the appropriate return value type. We sought a way to achieve this without dynamic dispatch overhead. This is more complex to do than with entry point versioning, because a given return statement can potentially jump back to many different call sites. The solution we have found uses speculation and deoptimization to propagate return value types only for functions which always return values of the same type at run time.

7.3 Alternative Solutions

Baptiste & Feeley have investigated the use of a mechanism to propagate function argument types based on dynamic dispatch in their versioning compiler for Scheme [33]. In their system, closures store a pointer to a table of entry point addresses. The table is indexed based on the type signature at the call site. This system makes it possible to pass argument type information when the identity of the callee is not known, which is

currently necessary in their system because they do not have an equivalent to our typed shapes mechanism. The main downside is that each call requires two indirections in order to get an entry point address, whereas in our system, a direct jump can be used. However, their system could conceivably perform better for megamorphic call sites.

The strategy we use to propagate return value types is speculative. Instead, we could have used a mechanism based on dynamic dispatch, dynamically choosing a version of the call continuation corresponding to the returned value type. However, we saw the use of dynamic dispatch as undesirable because the added cost of dynamic dispatch is likely as much or more as the cost of simply testing the return value type, which is one single dynamic type test.

Discussions with a compiler engineer from the Google V8 team revealed that V8 does not currently perform interprocedural type analysis of any sort, as such analyses are considered too costly. Instead, V8 relies on inlining of frequent calls to eliminate type checks interprocedurally in hot methods. The techniques we have developed could likely help improve the performance of modern JS VMs, particularly in recursive code.

7.4 Results

With interprocedural propagation of type information, BBV becomes competitive, in terms of optimization capabilities, with whole-program type analysis. One of the most important results given in the following article is that we are able to eliminate 94.3% of dynamic type checks on average, across our set of benchmarks. We are able to show that this is more than what is achievable through static whole-program type analysis alone. This somewhat counter-intuitive result demonstrates that BBV is not merely a quick approximation of a more traditional type analysis, as it can actually be more precise in practice.

Interprocedural Type Specialization of JavaScript Programs Without Type Analysis

Maxime Chevalier-Boisvert[1] and Marc Feeley[2]

1 **DIRO, Université de Montréal**
Montreal, Quebec, Canada
2 **DIRO, Université de Montréal**
Montreal, Quebec, Canada

Abstract

Previous work proposed lazy basic block versioning, a technique for just-in-time compilation of dynamic languages which we believe represents an interesting point in the design space. Basic block versioning is simple to implement, simple enough that a single developer can build a complete just-in-time compiler for JavaScript in a year, yet it performs surprisingly well as it propagates context-sensitive type information to generate type-specialized code on the fly.

In this paper, we demonstrate that lazy basic block versioning can be extended in simple ways to propagate type information across function call boundaries. This gives some of the benefits of whole-program analysis, or a tracing compiler, without having to implement the machinery for either. We have implemented this proposal in the Higgs JavaScript virtual machine and report on the empirical evaluation of this system on a set of industry standard benchmarks. The approach eliminates 94.3% of dynamic type tests on average, which we show is more than what is achievable with any static whole-program type analysis.

1998 ACM Subject Classification D.3.4 Programming Languages: Processors—compilers, optimization, code generation, run-time environments

Keywords and phrases Just-In-Time Compilation, Dynamic Language, Optimization, Object Oriented, JavaScript

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

A production compiler for a widely used dynamic language such as JavaScript is an intricate piece of software, usually the outcome of 10 to 100 developer-years of effort. The architecture of such a compiler is one of the first design decisions made during development. This decision is rarely revisited, as architectural changes tend to be disruptive. In previous work, Chevalier-Boisvert and Feeley argued for an architecture based on the concept of lazy Basic Block Versioning (BBV) [14]. They claimed that the technique hits a sweet spot in the tradeoff between implementation complexity and performance of the generated code. As evidence they designed and implemented Higgs, a JavaScript virtual machine and Just-In-Time (JIT) compiler which has performance competitive with other research virtual machines and can sometimes match the performance of production systems such as V8. Notably, the Higgs compiler took about a year of development time. The reduced development time is particularly important for languages that are maintained by small teams of volunteers. Lazy BBV occupies a point in the design space of JIT compilers that is between method-based compilers and tracing JITs such as Mozilla's TraceMonkey [17], and run-time specialization of Oracle's Truffle [36]. The simplicity of BBV is one of its main advantages. It does not

require additional infrastructure such as a static analyzer to approximate program facts, or an interpreter to record traces.

BBV is a simple and elegant compilation technique to optimize dynamically typed programs on the fly. The technique uses dynamic type tests which are part of the implicit semantics of primitive operators in dynamically typed languages to capture and propagate type information. Type-specialized versions of individual basic blocks are lazily compiled based on the types encountered during the execution of programs. The technique, as described in [14], is limited to optimizing type checks on local variables within a single function. The compiler has no information on the types of arguments, return values, or object properties, and is thus unable to eliminate some redundant dynamic type checks.

This paper extends basic block versioning with the ability to propagate type information across function call boundaries and to specialize code based on the type of object properties. In the framework of basic block versioning, these extensions are easy to implement and seem to work rather well. This paper makes the following specific contributions:

1. The combination of BBV with a *typed object shape* mechanism which encodes property type information including *method identity*, enabling the compiler to know the identity of callees at call sites (Section 4.1).
2. The extension of BBV with *specialized function entry points*, which makes it possible to pass argument types from callers to callees. This is done efficiently, without dynamic dispatch, using method identity information provided by typed object shapes (Section 4.2).
3. A speculative technique for *call continuation specialization*, which enables type information about return values to be passed from callees back to callers, without dynamic overhead (Section 4.3).

To validate our claims we implemented these contributions in the Higgs JavaScript compiler and evaluated its performance on industry standard benchmarks (Section 5).

A word about evaluation is in order. We considered implementing our ideas within an existing JavaScript compiler, but quickly realized that the architectural changes required were beyond our resources. Thus we picked Higgs as a vehicle for our experiments. This choice comes at a cost; comparing performance of a research prototype to a production system is tricky. A production system has a mature garbage collector, highly tuned libraries, and performs a massive number of optimizations (many, but not all, of which are orthogonal to this work). A research prototype is likely to not have any of those. It is thus not surprising that Higgs runs roughly half as fast as V8. This may be a sign that our approach is inherently limited, or that we simply lack the resources of major corporations. Cognizant of the inherent limitations of empirical evaluations, we have chosen the following approach. We measure the improvement of the techniques presented in this paper by the number of type tests we are able to eliminate and the performance impact over the previous version of the Higgs compiler. This gives us a metric of progress. We compare our implementation with two relevant systems, one is the TraceMonkey tracing compiler. The reason for this comparison is that basic block versioning has been compared by others to tracing compilation. It is thus interesting to see how the two perform on the same benchmarks. Then we choose Truffle/JS as an example of a research prototype, albeit one implemented by a large team of industrial researchers. For completeness we include, in Appendix A, performance results comparing Higgs to leading commercial JavaScript implementations.

2 Influences and Related Work

The literature on just-in-time compilation is rich with, by now, decades of work. The work presented here was influenced by many results obtained in the Self project and should be contrasted to work on type analysis and dynamic compilation of dynamic languages.

Shapes. The notion of describing objects with shapes can be traced back to the Self programming language [11, 22], where so-called maps group objects cloned from the same prototype. Like shapes, maps reduce memory usage and stored metadata relating to properties (though not type information). Today, commercial JavaScript implementations such as V8, SpiderMonkey, Nitro and Truffle/JS have all adopted this idea. Each object contains a pointer to its shape, which describes the layout of the object and property attribute metadata. Truffle introduced the notion of specializing shapes based on property types to the literature [35]. This paper builds on that idea and demonstrates how to effectively integrate such a model with basic block versioning.

Splitting. Basic block versioning bears resemblance to Self's *iterative type analysis* and *extended message splitting* [13] which combines static analysis with a transformation that compiles multiple versions of loops and duplicates control flow paths to eliminate type tests. The analysis works in an iterative fashion, transforming the control flow graph of a function while performing a type analysis. It integrates a mechanism to generate new versions of loops when needed, and a message splitting algorithm to try and minimize type information lost through control flow merges. One key disadvantage is that statically cloning code requires being conservative, generating potentially more code than necessary, as it is impossible to statically determine exactly which control flow paths will be taken at run time, and this must be overapproximated. The approach also has roots in Agesen's cartesian product algorithm [2] which avoids the loss of type information at control-flow merges by representing program state with sets of vectors of concrete types.

Analysis. There have been multiple efforts to devise type analyses for dynamic languages. Rapid Atomic Type Analysis [27] is an intraprocedural flow-sensitive analysis that assigns unique types to each variable. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [5], Ruby [16] and Python [4], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [23, 24, 25]. Such analyses are too time consuming to be used in a just-in-time compiler. Kedlaya, Roesch et al. [26] improved the precision of type analyses by combining them with type feedback and profiling. This shows promise, but does not deal with object shapes and property types. Work has also been done on a flow-sensitive alias analysis for dynamic languages [19], but it is still unclear if the analysis can be used on-line. More recently, Brian Hackett et al. presented an interprocedural hybrid type analysis for JavaScript suitable for use in a just-in-time compiler [21]. While this is an important step forward, it remains vulnerable to imprecise type information polluting analysis results. Basic block versioning can help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Tracing. Trace compilation, introduced by Dynamo [6] and later applied to just-in-time compilation in HotpathVM [18], aims to record sequences of instructions executed inside hot loops. Such sequences make optimization simpler. Type information is accumulated along traces and used to specialize code and remove type tests [17], overflow checks [34]

or unnecessary allocations [8]. Basic block versioning resembles tracing in that context updating works on essentially linear code fragments and code is optimized similarly to what may be done in a tracing compiler. Code is also compiled lazily, as needed, without compiling whole functions at once. Trace compilation [9] and meta-tracing are an active area of research [10]. The simplicity of basic block versioning is one of its main advantages. It does not require external infrastructure such as an interpreter to record traces. Trace compiler implementations must deal with corner cases that do not appear with basic block versioning. With trace compilation, there is the potential for trace explosion if there are a large number of control flow paths going through a loop [7]. It is also not obvious how many times a loop should be recorded or unrolled to maximize the elimination of type checks. This problem is solved with basic block versioning since versioning is driven by type information and there is a natural bound to the number of versions that comes from the finite number of types in the system. Trace compilers must implement parameterizable policies and mechanisms to deal with recursion, nested loops and potentially very long traces that do not fit in instruction caches.

Customization. Customization is another technique developed to optimize Self programs [12]. It compiles multiple copies of methods specialized on the receiver object type. Similarly, *type-directed cloning* [28] clones methods based on argument types, producing more specialized code using richer type information. The work of Chevalier-Boisvert et al. on just-in-time specialization for MATLAB [15] and similar work done for the MaJIC MATLAB compiler [3] tries to capture argument types to dynamically compile optimized versions of whole functions. All of these techniques are forms of type-driven code duplication aimed at extracting type information. Basic block versioning operates at a lower level of granularity, allowing it to find optimization opportunities inside of method bodies by duplicating code paths. There are similarities between the Psyco JIT specialization work and our own. The Psyco prototype for Python [31] interleaves execution and JIT compilation to gather run time information about values. It then specializes code on the fly based on types and values. It also incorporates a scheme where functions can have multiple entry points. We extend upon this work by combining a similar approach, that of basic block versioning, with typed shapes and a mechanism for propagating return types from callees to callers with low overhead. The *tracelet-based* approach used by Facebook's *HHVM* for PHP [1] bears similarities to our own. HHVM compiles small code regions (tracelets) which are single-entry multiple-exit basic blocks. Each tracelet is type-specialized based on variable types observed at compilation time. Guards are inserted at the entry of tracelets to verify at run time that the types observed are still valid for all future executions. High-level instructions in tracelets are specialized based on the guarded types. If these guards fail, new versions of tracelets are compiled based on different type information and chained to the failing guards. One difference with our work is that HHVM uses an ahead-of-time type analysis pass. Another difference is that with the approach described in [1], each tracelet re-checks the types of its inputs, whereas BBV propagates known types to successor blocks and doesn't usually need to re-check the types of local variables. Finally, HHVM falls back on an interpreter when too many tracelet versions are generated. Higgs falls back to generic basic block versions which do not make type assumptions but are still compiled. Beyond type specialization, recent work by Costa et al. on *just-in-time value specialization* has shown that specializing JavaScript functions based on specific argument values can lead to performance improvements [33], as many functions are always called with the same arguments.

3 Background

The work presented in this paper is implemented in a research virtual machine for JavaScript (ECMAScript 5) known as Higgs [1]. The Higgs virtual machine includes a just-in-time compiler built around lazy basic block versioning. This compiler is intended to be lightweight with a simple implementation. Code generation and type specialization are performed in a single pass. Register allocation is done using a greedy allocator. The runtime and standard libraries are self-hosted, written in an extended dialect of JavaScript with low-level primitives. These low-level primitives are special instructions which allow expressing type tests, pointer manipulation, as well as integer and floating point machine instructions in the source language.

3.1 Value Types and Type Tests

Higgs segregates values into categories based on type tags [20]. These type tags form a simple type system that is used for versioning. The types are mostly straightforward and correspond closely to values manipulated by JavaScript programs. The one exception is the `unknown` type tag that is used by the compiler to indicate that no information is available for the corresponding value.

`int32`	signed 32-bit integers
`float64`	64-bit floating point numbers
`undef`	the `undefined` value
`null`	the `null` value
`bool`	`true` and `false` boolean values
`string`	strings
`array`	arrays
`closure`	function objects
`object`	Plain JS objects
`unknown`	type unknown

JS is a dynamically typed and late-bound programming language. There are no static type annotations, and the types of variables may change during the execution of a program. As such, there are many implicit type checks hiding in even the simplest JS programs. Figure 1 shows an iterative function which illustrates this. The `sum` function contains three primitive operators: a comparison, a decrementation and an addition. Each of these operators implicitly checks the types of its operands as part of its semantics.

In all, there are four implicit type checks hiding in the `sum` function:

1. The `>` operator checks the type of `n` before comparing it against the integer zero.
2. The type of `s` is checked before computing `s += i`
3. The type of `i` is also checked before computing `s += i`
4. The decrementation operator checks the type of `i` before computing `--i`

A naive JS implementation performs these type checks every time an operator is evaluated. In Higgs, this is done using primitive instructions which can test the type tags of values. Figure 2 illustrates the primitive operations and implicit type tag checks executed by Higgs with basic

[1] https://github.com/higgsjs

```
function sum(n) {
    var s = 0;
    for (var i = n; i > 0; --i)
        s += i;
    return s;
}

sum(500);
```

Figure 1 Iterative JS `sum` function.

block versioning disabled when `sum(500)` is evaluated. When computing `sum(500)`, only small integer (`int32`) values are used, and so, much of these type checks are redundant.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I  // if not (n > 0)

C: if not is_int32(s) goto stub2
D: if not is_int32(i) goto stub3
E: s = add_int32(s,i)
   if overflow goto stub4

F: if not is_int32(i) goto stub5
G: i = sub_int32(i,1)
   if overflow goto stub6
H: goto B

I: return s
```

Figure 2 Control-flow graph of the `sum` function before BBV

The `is_int32` primitives act as guards which verify that the type tag associated with a given variable is `int32` before executing a machine instruction specific to integer values. Should any of these tests fail, execution will flow to a stub that generates new machine code to handle non-integer values. The overflow test primitives serve to verify that an integer overflow did not occur, and handle such an occurrence otherwise.

3.2 Lazy Basic Block Versioning

Basic block versioning is a just-in-time code generation technique originally applied to JavaScript by Chevalier-Boisvert & Feeley [14], and adapted to Scheme by Saleil & Feeley [32]. The technique bears similarities to HHVM's tracelet-based compilation approach and Psyco's just-in-time code specialization system [31].

BBV works at the level of individual basic blocks. We define a basic block as a single-entry single-exit sequence of instructions. Basic blocks end with one branching instruction which jumps to other basic blocks. In Higgs, basic blocks are usually short, sometimes just one instruction in our Intermediate Representation (IR), due to the large number of type tests, each of which is treated as a branching instruction which terminates the current basic block.

The BBV engine interleaves compilation and execution. It generates machine code for basic blocks lazily, instantiating them into one or more versions, each type-specialized based on accumulated type information. BBV propagates type information by maintaining a context for each block version which stores known type information about live variables.

This context is updated as block versions are compiled.

Type tag tests are used to capture type information and enrich the versioning context. We know, for instance, that if we branch on the "true" side of an `is_int32(n)` test, then `n` must have tag `int32` in the successor block. This fact is exploited by instantiating a specialized version of the successor block based on the knowledge that `n` is `int32`. Because BBV uses lazy code generation, it never generates block versions for types that do not occur at run time. It achieves this by delaying the compilation of conditional branch targets using machine code stubs.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I

// s,i,n are known to be int32
C: //is_int32(s) check eliminated
D: //is_int32(i) check eliminated
E: s = add_int32(s,i)
   if overflow goto stub2

// s,i,n are known to be int32
F: //is_int32(i) check eliminated
G: i = sub_int32(i,1)
   if overflow goto stub3
H: goto B

I: return s
```

Figure 3 Control-flow graph of the `sum` function after BBV

Using BBV, three of the four implicit type checks in the `sum` function from Figure 1 are eliminated. The resulting optimized control flow graph is shown in Figure 3. A single type test remains: the type of `n` is tested when entering the function. When first executing the `sum(500)` call, Higgs takes the following steps to compile and optimize the `sum` function:

- The `sum` function is entered, block `A` is executed. The `s` and `i` variables are initialized to `0`. The context is updated to indicate both `s` and `i` have type tag `int32`. The type of `n` is unknown. The `is_int32(n)` branch is made to point to machine code stubs and execution is resumed.
- Execution resumes. The `is_int32(n)` check evaluates to "true". A stub for block `B` is hit. This stub calls back into the compiler.
- Compilation resumes, and a version of block `B` with `n` known to be `int32` is generated. Stubs are generated for the `gt_int32(n,0)` branch targets.
- Execution resumes. A stub of block `C` is hit.
- Compilation resumes. A version of block `C` with `n` known to be `int32` is produced. The variables `s` and `i` are already known to be `int32`, hence the type tag checks in `C` and `D` can be evaluated at compilation time and eliminated. A stub is produced for the integer overflow check.
- Execution resumes. No overflow occurs, a stub for block `F` is hit.
- Compilation resumes. A version of block `F` with `s`, `i` and `n` as `int32` is compiled. The type check in `F` is evaluated at compilation time and eliminated. Stubs for the overflow branch in `G` are produced.
- Execution resumes. No overflow occurs, a stub of block `H` is hit.

- Compilation resumes. Block `H` produces a jump to the version of `B` that was already generated, where `s`, `i` and `n` are all known to be `int32`.
- Execution resumes and continues until the `gt_int32(n,0)` test in block `B` fails. Note that no more type checks are executed.
- The loop test fails. A stub for block I is hit. Block `I` is compiled.
- Execution resumes at block `I`, the `sum` function returns to the caller.

Because of its JIT nature, BBV has at least two powerful advantages over traditional static type analyses. The first is that BBV considers only the parts of the control flow graph that get executed, and it knows precisely which they are, as machine code is only generated for basic blocks which are executed. The second is that code paths can often be duplicated and specialized based on different type combinations, making it possible to avoid the loss of precision caused by control flow merges in traditional type analyses.

3.3 Motivating Example

The example in Figure 1 is one for which plain intraprocedural BBV works particularly well. In this section, we will provide a motivating example for our work which highlights the limitations of the unextended BBV approach described in [14]. We will then show how we have extended BBV to remove these limitations.

```
function sumList(lst)
{
    if (lst == null)
        return 0
    return lst.val + sumList(lst.next)
}

function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect␣sum')
```

Figure 4 JS function to recursively sum the values stored in a linked list.

Figure 4 shows the `sumList` function for recursively traversing a linked list and computing the sum of numerical values stored in each node. While this small program may appear simplistic, there is much semantic complexity hidden behind the scenes. A correct but naive implementation of this function contains six implicit dynamic type tag tests, which must be eliminated to maximize performance:

1. The tag of the `lst` argument is checked when comparing it against `null`.
2. The tag of `lst` is re-checked before reading the `lst.val` property.
3. The tag of `lst` is checked a third time before reading the `lst.next` property.
4. The `sumList` function is a mutable global variable. Before calling it, there is an implicit check to make sure that this is in fact a closure.
5. The tag of `lst.val` is checked before computing `lst.val + sumList(lst.next)`.
6. The tag of `sumList(lst.next)` is also checked, because functions calls can return values of any type.

The BBV algorithm described in [14] is limited to an intraprocedural scope, that is, it deals with local variable types only. It cannot pass type information between callers and callees. It also assumes that all object properties (including global variables, which are properties of the global object) have unknown type. As such, the unextended BBV algorithm is ill-equipped to optimize the `sumList` function, or object-oriented JS code in general.

The implicit tests executed by a version of Higgs without BBV are shown in Figure 5.

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: val = read_prop(lst, 'val')
   if not is_object(lst) goto stub2

D: next = read_prop(lst, 'next')
   sumfn = read_prop(globalObj, 'sumList')
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if overflow goto stub6

H: return t2

I: return 0
```

Figure 5 Implicit type checks in the `sumList` function.

Once the type tag of the `lst` parameter has been tested and found to be `object`, intraprocedural BBV can eliminate the second `is_object` test. Unfortunately, it cannot eliminate any of the other type tag tests. Since nothing is known about object property types, the type tags of the `val` and `next` properties must be tested for each call. The type tag of `sumList` is also tested before every call. Lastly, the return type of the `sumList` call is checked after each call. Clearly, most of these checks are provably redundant, and it should be feasible to eliminate them. The next sections will explain the ways in which we have extended BBV to give it the necessary capabilities.

4 Interprocedural Basic Block Versioning

This section describes the three extensions to basic block versioning that allow us to propagate type information across procedure calls.

4.1 Typed Object Shapes

BBV, as presented in [14], deals with function parameter and local variable types only. It has no mechanism for attaching types to object properties. This is particularly problematic because, in JS, functions are typically stored in objects. This includes object methods and also global functions (JS stores global functions as properties of the *global object*). We would like to attach type tags to object properties, global variables included.

4.1.1 Object Shapes and Shape Tests

Currently, all commercial JS engines have a notion of object shapes, which is similar to the notion of property maps invented for the Self VM. That is, any given object contains a pointer to a shape descriptor providing its memory layout: the properties it contains, the property slot index (memory offset) each property is stored at, as well as attribute flags (i.e. writable, enumerable, etc.). For instance, linked list nodes and the global object in the example from Figure 4 have shapes `S` and `G`, shown in Figure 6.

```
// Linked list node shape
S: { val: slot 0, next: slot 1 }

// Global object shape
G: {
    ...,
    Error: slot 1,
    ...,
    makeList: slot 30,
    sumList: slot 33,
    lst: slot 34
}
```

Figure 6 Linked list node and global object shapes.

Traversing shape data structures on each object property access would be prohibitively expensive. As such, Higgs and all modern JS engines optimize property accesses using Polymorphic Inline Caches (PICs) [22]. PICs are lazily updated sequences of inlined machine instructions which implement property reads and writes. Typically, a cascade of conditional branch instructions establish the shape of an object in order to determine the memory offset at which the property to be read or written is stored. A specialized machine instruction is then executed which accesses the property at the correct offset. PICs are extended as needed to handle previously unseen object shapes.

In the `sumList` function, there are three property reads, and therefore three PICs. Linked list nodes and the global object only have one possible shape, and so there is only one shape test inside each PIC. The primitive operations and dynamic tests executed by an unextended implementation of Higgs which uses PICs are illustrated in Figure 7.

4.1.2 Extending Shapes with Types

Work done on the Truffle Object Model (OSM) [35] describes how object shapes can be straightforwardly extended to also encode type tags for object properties. Property writes are guarded to update object shapes when a property type changes. Property reads establish the shape of objects in order to know the memory offset at which to read properties. When object shapes also encode the type tags of properties, establishing the shape of an object tells us not only where to read the property, but also what type tag this property has. Hence, the cost of guarding property writes is easily offset, because typical JS programs have many more property reads than property writes. A small overhead is paid to guard property writes, and in exchange, type checks after property reads are effectively eliminated.

We extend upon the original BBV work with a *typed object shape* system inspired by the Truffle OSM. This model is a natural fit for the BBV algorithm. Our extended BBV

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S) call updatePIC  // PIC 1
   val = read_slot(lst, 0)                 // PIC 1
   if not is_object(lst) goto stub2

D: if not is_shape(lst, S) call updatePIC  // PIC 2
   next = read_slot(lst, 1)                // PIC 2
   if not is_shape(globalObj, G) call updatePIC // PIC 3
   sumfn = read_slot(globalObj, 33)            // PIC 3
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if not overflow goto stub6

H: return t2

I: return 0
```

Figure 7 Primitive operations in `sumList` executed by an unextended version of Higgs.

algorithm not only propagates known type tags associated with values, but also object shapes. The shape tests which are normally part of PICs allow our JIT compiler to establish and propagate the shape of an object in the same way that type tag tests enabled BBV to extract and propagate the type tags of values. Once the shape associated with an object is known to the BBV engine, then, by extension, the types of all properties read from that object are also known.

In order to enable interprocedural type propagation, it is useful to know which function is being called for as many call sites as possible, both for calls to global functions and method calls. As such, we have gone one step further than the Truffle OSM, and attached not only type tags to object shapes, but also *method identity information*. That is, for properties which have the closure type tag, shapes encode a pointer to the IR node corresponding to the function the property is a closure of. This enables us to know the identity of callees at code generation time for the large majority of call sites.

With typed shapes, linked list nodes from the `sumList` have two possible shapes, one where the `next` property is `null`, and one where it is an object. The global object encodes not only the offsets of global variables, but also the identity of global functions. This is illustrated in Figure 8.

In order to allow BBV to take advantage of typed shape information, we break up PICs into their component parts. PICs, which were previously monolithic sequences of inlined machine instructions, are now exposed in our compiler IR as separate shape test and memory access instructions. The result is that the regular BBV mechanisms can be leveraged to extract shape information from shape tests and propagate it. Propagating shape information (and the associated property types), allows us to optimize the `sumList` function as shown in Figure 9.

Two separate code paths are generated inside the `sumList` function, one for each of the two possible shapes of the linked list nodes. More code is generated, but on any given code

```
// Linked list node shape
S1: { val: (slot 0, int32), next: (slot 1, null) }
S2: { val: (slot 0, int32), next: (slot 1, object) }

// Global object shape
// Closures have method identity information
G: {
    ...,
    Error: (slot 1, closure/Error),
    ...,
    makeList: (slot 30, closure/makeList),
    sumList: (slot 33, closure/sumList),
    lst: (slot 34, object)
}
```

Figure 8 Typed object shapes encode property type information.

path, at most three type tag tests are executed instead of five. Since linked list nodes now have two possible shapes, we may test the shape of linked list nodes twice instead of just once when reading the `lst.val` property. However, because we no longer employ monolithic inline caches, this shape is propagated from the property read of `lst.val` to that of `lst.next`. Hence, as a result, we actually perform less dynamic shape tests on average.

4.2 Entry Point Versioning

Procedure cloning has been shown to be a viable optimization technique, both in ahead of time and JIT compilation contexts. By specializing function bodies based on argument types at call sites, it becomes possible to infer the types of a large proportion of local variables, allowing effective elimination of type checks.

Our first extension to BBV is to allow functions to have multiple type-specialized entry points. That is, when the identity of a callee at a given call site is known at compilation time, the JIT compiler requests a specialized version of the entry point block for the callee. This specialized entry point assumes the argument types known at the call site. Type information is thus propagated from the caller to the callee.

Inside the callee, BBV proceeds as described in [14], deducing local variable types and eliminating redundant type checks. Our approach places a hard limit on the number of versions that may be created for a given basic block, and so automatically limits the number of entry points that may be created for any given function. If there are already too many specialized entry points for a given callee, a generic entry point is obtained instead. This does not matter to the caller and occurs rarely in practice.

Propagating types from callers to callees allows eliminating redundant type tests in the callee, but also makes it possible to pass arguments without boxing them, thereby reducing the overhead incurred by function calls. Note that our approach does not use any dynamic dispatch to propagate type information from callers to callees. It relies on information obtained from typed shapes to give us the identity of callees (both global functions and object methods) for free. When the identity of a callee is unknown, a generic entry point is used.

In the case of the linked list example from Section 3.3, we can specialize the `sumList`

```
A: if is_null(lst) goto I:

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S1) goto C2
   val = read_slot(lst, 0) // val is known to be int32
   next = read_slot(lst, 1) // next is known to be null

D: if not is_shape(globalObj, G) goto stub2
   sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E: t1 = sumfn(next)
   if not is_int32(t1) goto stub3

G: t2 = add_int32(val, t1)
   if overflow goto stub4

H: return t2

I: return 0

C2: if not is_shape(lst, S2) goto stub5
    val = read_slot(lst, 0) // val is known to be int32
    next = read_slot(lst, 1) // next is known to be object

D2: if not is_shape(globalObj, G) goto stub6
    sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E2: t1 = sumfn(next)
    if not is_int32(t1) goto stub7

G2: t2 = add_int32(val, t1)
    if overflow goto stub8

H2: return t2
```

Figure 9 The `sumList` function optimized with typed shapes.

function entry point based on the type tag of the `lst` parameter. As a consequence, we know whether `lst` has tag `null` or `object` upon entering the function.

With entry point versioning, we can eliminate all type tag checks, except for the check on the return type of the `sumList` call. This test seems redundant, considering that, in our example, the `sumList` function only ever returns `int32` values. The following section will explain our strategy to optimize this.

4.3 Call Continuation Specialization

Achieving full interprocedural type propagation demands passing the return type information from callees to callers. While it is fairly straightforward to establish the identity of the callee a call site will jump to in the majority of cases, establishing where a `return` statement will jump to is less straightforward. This is to say, most call sites are monomorphic and jump to a single function, and hence, a single specialized entry point. Furthermore, versioning code based on object shapes has the net effect that it will often split polymorphic call sites into monomorphic ones, which is very convenient for us.

We would like to version call continuations (the code executed when we return from a call) in accorance with the return types observed during execution. However, one `return` statement can potentially jump to several call continuations within a program. This means we cannot employ the same strategy as with entry point versioning. We cannot simply jump

from one `return` statement to a specialized call continuation which assumes a known return type. Type information about return values could be propagated with a dynamic dispatch of the return address indexed with the result type. However this would incur a run time cost. We would be trading one form of dynamic overhead (that of type checks) for another (that of dynamic dispatch).

Instead, we have chosen to extend BBV with an approach that has zero run time cost (amortized overhead). Call continuations are compiled lazily when the first return to a given continuation is executed. When a function first executes a `return` statement, its return type, if known, is memorized. Call continuations are then speculatively optimized based on this memorized return type. If later returns from this function turn out to have a different return type, the optimized call continuations are invalidated (see Section 4.3.1).

```
function f()
{
    // Call site
    var r = g()

    // Call continuation
    // The addition has an implicit type check
    return r + 2
}

function g()
{
    return 1
}
```

Figure 10 Function call with a fixed return type.

Given the small example given in Figure 10 where a function `f` calls some function `g`, where `g` always returns values of type `int32`, the call continuation specialization process continuations takes the following steps:

- A call to `g` is encountered. Assuming the identity of the callee is known from typed shapes (otherwise this optimization is not performed), `f` is added to a list of callers of `g`.
- A stub is generated for the call continuation in `f`.
- Machine code for the call site is generated, it is made to jump directly to a specialized entry point in `g`.
- Execution resumes in `f` and jumps to `g`. Execution continues until `g` returns.
- Compilation resumes. The compiler has determined that `g` returns an `int32` value since the function `g` is annotated to indicate that it returns `int32` values.
- Execution resumes and `g` returns to the call continuation in `f`. The call continuation stub is executed.
- The call continuation in `f` is compiled. The compiler sees that `g` has been annotated as returning `int32` values. The code in `f` is optimized using this type information. No type check is performed at the addition.

The call continuation specialization process presented so far is able to optimize recursive calls in the `sumList` example and eliminate the type tag check on the return value. However, as explained in the next section, this process is speculative and does not work for every function.

4.3.1 Invalidating Call Continuations

The `makeList` function from Figure 4 is an example where the speculative call continuation specialization process fails. This is because `makeList` can return both objects and `null` values. As such, we cannot specialize callers of the function based on a single return type tag. In this situation, the speculative call continuation specialization process will try to specialize continuations, fail, and deoptimize them.

The first time that the `makeList` function returns, it will return a `null` value. This first return will then trigger the compilation of a specialized call continuation which assumes the return type of `makeList` to be `null`. When the function later returns a value with type tag `object`, this will be detected at code generation time. Callers of the `makeList` function will then have their call continuations deoptimized.

The deoptimization is done simply by writing stubs over already compiled call continuations. Should another `makeList` call return to a deoptimized call continuation, the stub will trigger the compilation of a new continuation. This time, the return type will not be specialized, because we know that `makeList` can return values with multiple type tags.

The speculative optimization and deoptimization process we employ could be seen as wasteful. We could have employed a static analysis instead. However, it can be difficult to establish the return type of a JS function simply by analyzing its code. Furthermore, the speculative approach can be more precise than a static analysis, because it is able to take the run time behavior of code into account. The `return` statements which are never executed will not be taken into account. A static analysis does not know exactly which `return` statements are executed and which are not, but BBV does.

5 Evaluation

This section reports on an empirical evaluation of interprocedural basic block versioning. This evaluation was carried out based on an implementation of the extensions presented in this paper, namely typed shapes, entry point specialization, and call continuation specialization, within the Higgs JavaScript compiler.

A total of 26 industry benchmarks were selected from the SunSpider and V8 suites. The authors decided not to use the JSBench benchmarks [29] as they are more suited to fast interpreters (they are short running and have little computation). Benchmarks for which performance hinges on compiling regular expressions were omitted, as this is not a feature supported by the Higgs compiler.

To measure steady state execution time separately from compilation time in a manner compatible with Higgs, V8, SunSpider, TraceMonkey, and Truffle/JS, the benchmarks were modified so that they could be run in a loop. Warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place. Unless otherwise specified, 1000 warmup iterations and 100 timing iterations are used.

V8 version 3.29.66, SpiderMonkey version C40.0a1, TraceMonkey version 1.8.5+ and Truffle/JS v0.9 were used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 CPU and 16GB of RAM running Ubuntu Linux 14.04. Dynamic CPU frequency scaling was disabled to ensure reliable timing measurements.

5.1 Method Identity

The extended version of Higgs tracks object shapes. Without them, the compiler would not be able to dermine which method is invoked at a call-site. With typed shapes, on average,

the identity of the callee method is known for 90% of calls executed dynamically. When entry point versioning and call continuation specialization are performed, that number increases to 97.5% of calls. In practice, the identity of callees is known for most call sites. The exceptions are dominated by implementation limitations of the current version of Higgs, which currently treats captured closure variables as having unknown type.

5.2 Type Tests

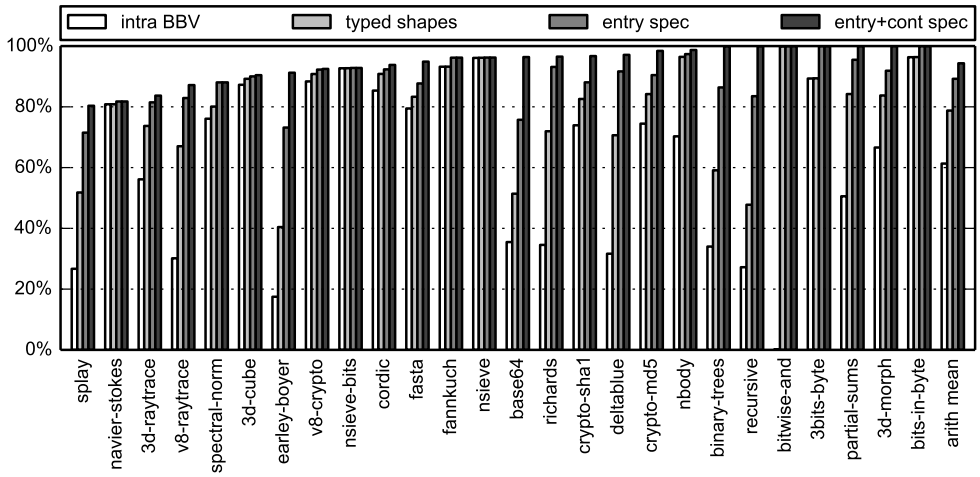

Figure 11 Proportion of type tests eliminated (higher is better).

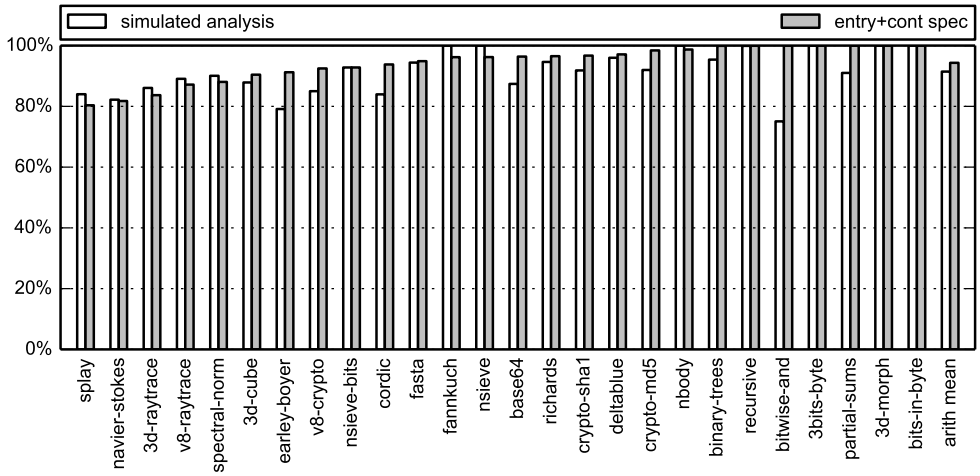

Figure 12 Proportion of type tests eliminated with BBV or a type analysis (higher is better).

Figure 11 shows the proportion of type tag tests eliminated with different variants of basic block versioning. These numbers measure actual tests executed at runtime rather than tests occuring in the program text. The first column (intra BBV) is the baseline, the number

of tests that could be eliminated with plain intraprocedural basic block versioning [14]. The second column (typed shapes) shows the results obtained by adding support for typed object shapes. The third column (entry spec) adds entry point specialization, and lastly, the fourth column (entry+cont spec) adds call continuation specialization. On average, the baseline eliminates 61% of tests, typed shapes increases this to 79%. Entry point specialization improves the result to 89%. Finally, the addition of call continuation specialization allows the elimination of 94.3% of dynamic tests, and, in several cases, nearly 100%.

5.3 Type Analysis

An obvious alternative to type propagation with interprocedural basic block versioning would be to perform a whole-program type analysis. As there are many different analyses in the literature with different degrees of precision, it is unclear how to evaluate the relative benefits of this paper's approach. It is possible to side-step the question by implementing an idealized static analysis. Each benchmark was run and the result of all tests was recorded. The benchmarks were then rerun with all type tests that always evaluate to the same result removed. The second run can be seen as an upper bound for the power of static analysis by itself. No static analysis can eliminate more tests than one that knows in advance the outcome of each of them. Figure 12 compares interprocedural basic block versioning and the idealized type analysis. The fact that basic block versioning outperforms type analysis should not come as a surprise. An analysis loses precision when control flow merges whereas basic block versioning creates separate versions to avoid this. The results suggest that no analysis can eliminate more than an average of 91.4% whereas Higgs can avoid executing 94.3% of tests. On more than half of the benchmarks, the proportion of eliminated tests exceeds 95%. In all benchmarks at least 80% of tests are removed.

5.4 Execution Time

The execution times of the benchmarks normalized to the unmodified version of the Higgs compiler [14] appear in Figure 13. With the exception of `navier-stokes`, `nsieve` and `nsieve-bits` (which are marginally slower), all benchmarks exhibit improvements. The largest speed up comes from the addition of typed object shapes, they improve execution time by an average of 26.8%. The addition of entry point specialization further improves performance, with a combined speedup of 36.3%. Finally, adding call continuation specialization brings the total improvement to 37.6%. The performance improvements brought by continuation specialization are relatively modest compared to those from entry point specialization. This is to be expected since entry point specialization allow us to eliminate more type tests (Section 5.2).

5.5 Shape Tests

Our implementation of typed shapes is able to propagate known object shapes from one property access to another. There are many instances where multiple property reads on the same object occur within a given function, and shape propagation can allow eliminating further shape tests after the first property access on an object. Enabling typed shapes results in an average decrease of 27% in the number of shape tests over an unextended implementation of Higgs which uses untyped shapes and inline caches.

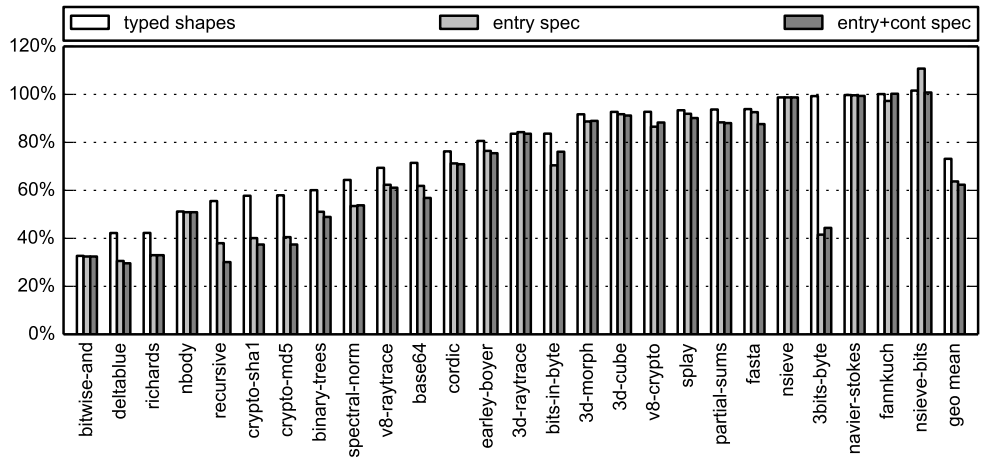

Figure 13 Execution time relative to baseline (lower is better).

5.6 Call Continuation Specialization

Call continuation specialization uses a speculative strategy to propagate return type information. Call continuations for a given callee may be recompiled and deoptimized if values are returned which do not match previously encountered return types. Empirically, only 2% of functions executed cause the invalidation of call continuation code. Dynamically, the type tag of return values is successfully propagated and known to the caller 72% of the time. In over half of the benchmarks, the type tag of return values is known over 99% of the time.

5.7 Code Size and Compile Time

Adding entry point and continuation specialization to the unmodified Higgs compiler cause an increase in generated machine code size of 5.5% in the worst case and just 1.0% on average. Intuitively, one may have expected a bigger code size increase given that entry point versioning can generate multiple entry points per function. However, better optimized machine code tends to be more compact. Compile time increases by 3.7% in the worst case and just 0.01% on average.

5.8 Tracing Compilation

Tracing compilation bears important similarities to basic block versioning. One could expect tracing to do better because it can optimize long linear sequences of code. Tracing compilation was introduced to JavaScript with the TraceMonkey [17, 34] compiler. This compiler was in production within Mozilla's browser until 2011. Figure 14 compares the performance of the two compilers. On average, Higgs is 2.7x faster than TraceMonkey, and performs better on 22 out of 26 benchmarks. The benchmarks TraceMonkey achieves the best performance on tend to be ones which feature short and predictable loops.

The difference between the two is striking. It should be noted that TraceMonkey was built by a considerably larger team and implements strictly more optimizations than Higgs. For instance, it can inline while recording a trace. Even without inlining, Higgs does much better on the largest benchmarks. The two raytrace benchmarks, for instance, make significant use

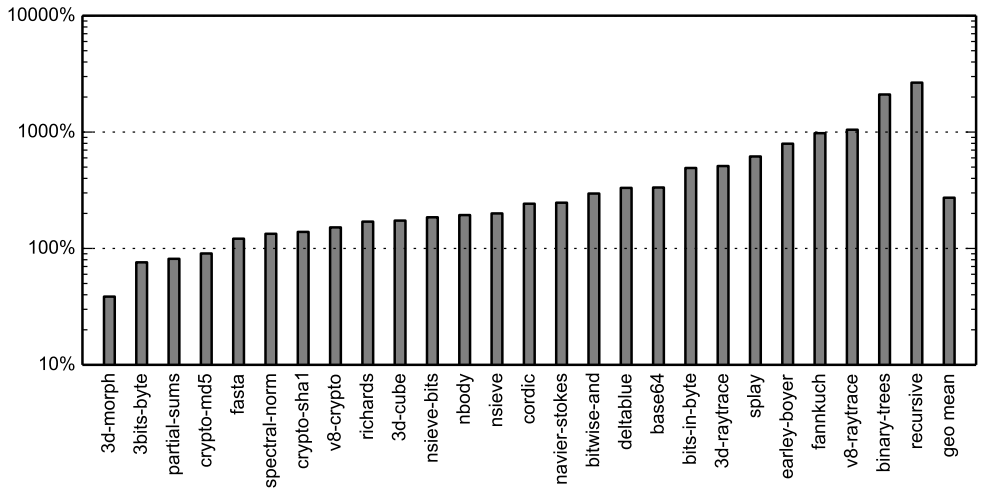

Figure 14 Speed relative to TraceMonkey (log scale, higher favors Higgs).

of object-oriented polymorphism and feature highly unpredictable conditional branches. The `earley-boyer` benchmark is the largest of all and features complex control-flow. The `splay` and `binary-trees` benchmarks apply recursive operations to tree data structures. We note that Higgs performs much better than TraceMonkey on the `recursive` microbenchmark which suggests TraceMonkey handles recursion poorly. While we caution against drawing definitive conclusions, it does appear that tracing compilation in the form implemented by TraceMonkey is mostly beneficial for computation with hot and predictable loops. Whereas Higgs is agnostic to the vagaries of control flow. It is worth mentioning that independent analysis of the behavior of real-world JavaScript programs suggests that hot and predictable loops are rare [30] and that TraceMonkey does not speed up real-world JavaScript programs such as the Google website [29].

5.9 Truffle/JS

Another interesting comparison is to look at the Truffle system from Oracle labs. Truffle/JS is an implementation of JavaScript written in Java and running on a modified Java virtual machine. Like Higgs, Truffle is a research prototype, but one being built by a larger team and with a code base about 6 times larger than Higgs'. It benefits from optimizations that are lacking in Higgs, such as method inlining and a sophisticated register allocator. For memory management it can defer to Java's highly tuned garbage collector.

Figure 15 shows the results of a performance comparison of Higgs against Truffle/JS. After both systems have gone through 1000 warmup iterations, Higgs is on average 69% as fast as Truffle/JS. The time recorded on the `3bits-byte` benchmark is zero, suggesting that Truffle used side effect analysis to optimize-away the computation.

Higgs and Truffle/JS, being research virtual machines, were not optimized for fast compilation. As a result, both systems are much slower than other engines when it comes to compilation times. We cannot directly measure the compilation time taken by Truffle/JS, but we can use the time it takes to warm up as a rough approximation.

Figure 16 shows the speed of Higgs relative to Truffle/JS when measuring the total time taken for 1000 iterations of our benchmarks, with no separate warmup iterations. On

average, Higgs is 220% as fast as Truffle/JS on this comparison, indicating that the warmup and compilation time for Higgs is much shorter. This is not surprising, since Higgs begins generating type-specialized machine code as soon as program execution begins.

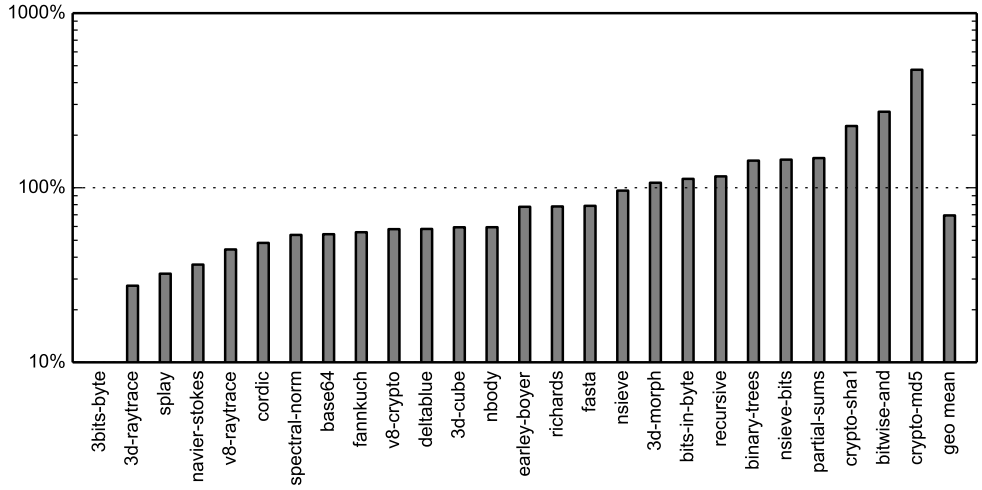

Figure 15 Speed relative to Truffle/JS (log scale, higher favors Higgs).

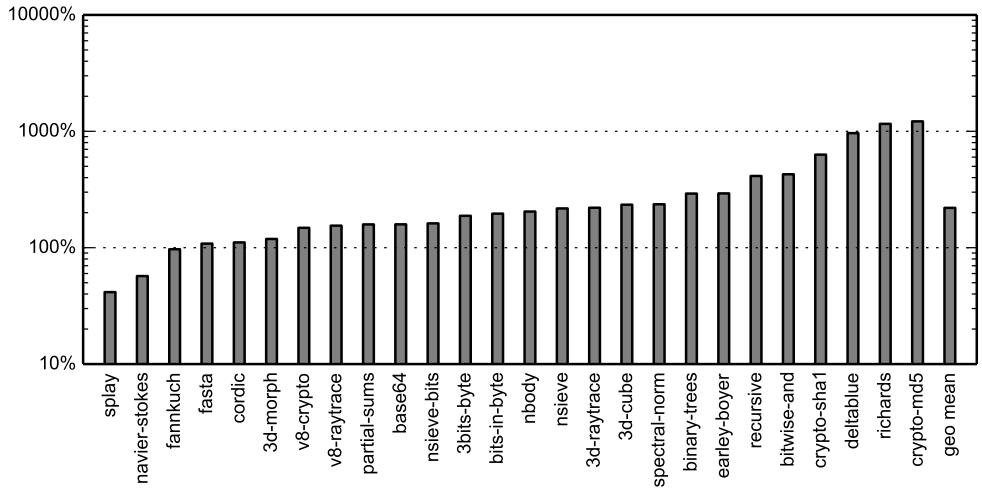

Figure 16 Speed relative to Truffle/JS, no warmup iterations (log scale, higher favors Higgs).

6 Conclusion

Basic block versioning is a compilation strategy for generating type-specialized machine code on the fly. This paper demonstrates how to extend this technique to propagate information across method call boundaries, both from callers to callees and from callees to callers, without requiring dynamic dispatch and without a separate type analysis pass.

Across 26 JavaScript benchmarks, interprocedural basic block versioning eliminates, on average, 94.3% of type tests. This is more than a static type analysis with access to perfect information could achieve. The proposed extension provides an average execution time reduction of 37.6% over an unextended basic block versioning implementation.

There is room for future work. While interprocedural basic block versioning yields encouraging results, more could be done. Two extensions to basic block versioning are planned: tracking types of closure variables and tracking array types. The Higgs compiler itself currently lacks several optimizations used by commercial virtual machines. While they are orthogonal to this paper, these optimizations may close the performance gap with commercial systems. The first optimization to add is method inlining. Inlining is likely synergistic with basic block versioning as it provides more contextual information but it runs the risk of increasing code size as versions proliferate. Bloat can be mitigated by lazy, incremental, inlining where basic blocks are only added when needed. This would be faster than inlining entire control flow graphs without needing recompilation of the entire caller at inlining-time.

Acknowledgements

Special thanks go to Laurie Hendren, Jan Vitek, Erick Lavoie, Vincent Foley, Paul Khuong, Molly Everett, Brett Fraley and all those who have contributed to the development of Higgs.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

References

1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *Proceedings of the 2014 conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 777–790. ACM New York, 2014.

2 Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

3 George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI)*, pages 294–303. ACM New York, May 2002.

4 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Dynamic Languages Symposium (DLS)*, pages 53–64. ACM New York, 2007.

5 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP 2005*, pages 428–452. Springer Berlin Heidelberg, 2005.

6 V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 conference on Programming*, pages 1–12. ACM New York, 2000.

7 Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

8 Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings*

of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM), pages 43–52. ACM New York, 2011.

9 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin 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*, pages 18–25. ACM, 2009.

10 Carl Friedrich Bolz, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. Meta-tracing makes a fast Racket. *Workshop on Dynamic Languages and Applications*, 2014.

11 C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, September 1989.

12 Craig Chambers and David Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI)*, pages 146–160. ACM New York, June 1989.

13 Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the 1990 conference on Programming Language Design and Implementation (PLDI)*, pages 150–164. ACM New York, 1990.

14 Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101–123. Schloss Dagstuhl, 2015. `http://arxiv.org/abs/1411.0352`.

15 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 46–65. Springer Berlin Heidelberg, 2010.

16 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1859–1866. ACM New York, 2009.

17 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

18 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE)*, pages 144–153. ACM New York, 2006.

19 Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM.

20 David Gudeman. Representing type information in dynamically typed languages, 1993.

21 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM New York, June 2012.

22 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

23 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

24 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings 17th International Static Analysis Symposium (SAS)*. Springer Berlin Heidelberg, September 2010.

25 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 2013 Dynamic Languages Symposium (DLS*. ACM New York, 2013.

26 Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. *SIGPLAN Not.*, 49(2):37–48, October 2013.

27 Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 66–83. Springer Berlin Heidelberg, 2010.

28 John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC)*, pages 566–580, 1995.

29 Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 677–694, 2011.

30 Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, June 2010.

31 Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM.

32 Baptiste Saleil and Marc Feeley. Code versioning and extremely lazy compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

33 Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

34 Rodrigo Sol, Christophe Guillon, FernandoMagno Quintão Pereira, and Mariza A.S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In Jens Knoop, editor, *Proceedings of the 2011 international conference on Compiler Construction (CC)*, pages 2–21. Springer Berlin Heidelberg, 2011.

35 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 133–144, New York, NY, USA, 2014. ACM.

36 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 2012 Dynamic Language Symposium (DLS)*, pages 73–82. ACM New York, 2012.

A Comparison with V8 and SpiderMonkey

Figure 17 compares the speed of Higgs to optimized commercial JavaScript virtual machines. Higgs is generally slower, sometimes by an order of magnitude. There are a few benchmarks where it outperforms V8. Notably, `bits-in-byte` features many function calls, and Higgs is able to optimize this fairly well. The `bitwise-and` microbenchmark is also interesting because it is a loop performing global object property accesses. Higgs outperforms every JS engine we have tested on this benchmark, suggesting that it has faster global property accesses, thanks to typed shapes. On the other hand, Higgs is slower everywhere else. This is probably because Higgs lacks orthogonal optimizations such as: loop-invariant code motion, global value numbering, bounds check elimination, automatic SIMD vectorization, method inlining, allocation sinking, floating-point register allocation, etc. In the absence of these optimizations, BBV is most promising for use in a baseline JIT compiler.

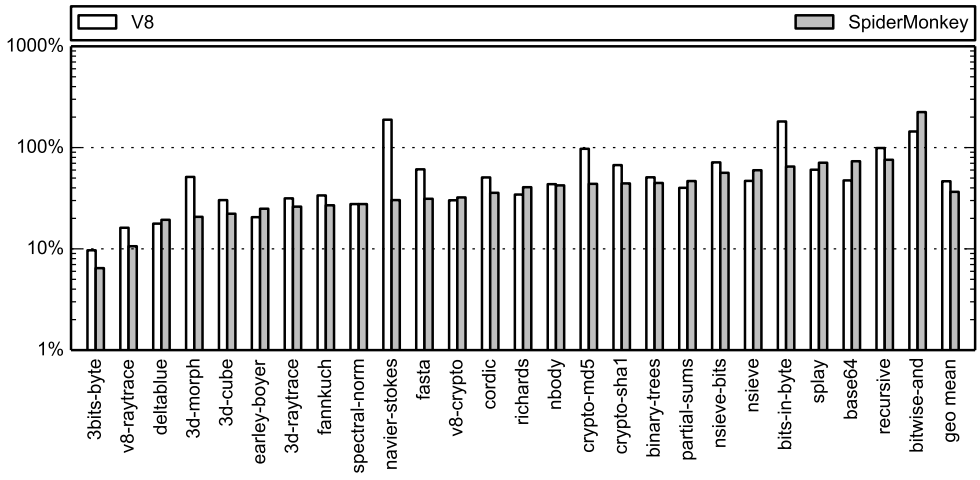

Figure 17 Speed of relative to commercial JS engines (log scale, higher favors Higgs)

CHAPTER 8

ADDITIONAL EXPERIMENTS

This chapter presents a few experiments which were not yet included in publications, but constitute interesting supplemental material. Chapters 4 to 7 demonstrate that BBV, despite its conceptual simplicity, is a very powerful technique for eliminating redundant dynamic type checks. Sections 8.1, 8.2 and 8.3 present three experiments which begin to explore the use of BBV for applications other than type-specialization. Section 8.4 is an investigation of the effect of eager and lazy generation of block versions on instruction cache misses and branch mispredictions. Finally, Section 8.5 looks at the performance of Higgs and commercial JS engines on a few microbenchmarks.

8.1 Overflow Check Elimination

8.1.1 Problem Description

Most JavaScript implementations use different representations for integer and floating-point values. The result of this is that operations such as integer addition and multiplication can potentially overflow, in which case the operands must be converted to floating-point values. To detect overflows, compilers must insert overflow checks in the generated code, which incur some code size and execution time penalty.

In the general case, eliminating overflow checks on arithmetic operations is difficult to achieve. This is because doing so requires having access to information about the range of numerical values being operated on, so that we can prove that the result of an addition, for instance, will remain within the range of integers the system can represent. This information is typically not available for most variables.

It occurred to us, however, that there is a common usage pattern in JavaScript where overflow checks are not so difficult to eliminate. This is the case where an integer index value is being incremented in a `for` loop, an example of which is given in Figure 8.1 The key element in such cases is that the bounds check condition `i < arr.length`

implies that `i` can be incremented by one without causing an overflow. This is necessarily the case, because the array length must be an integer (`int32`). Hence, when this condition is true, there is necessarily at least one representable integer greater than `i`, meaning the `i++` operation cannot overflow.

```
function arraySum(arr)
{
    var sum = 0;

    for (var i = 0; i < arr.length; ++i)
        sum += arr[i];

    return sum;
}
```

Figure 8.1: JavaScript `for` loop with integer index

Eliminating overflow checks in such loops may appear trivial, but there are multiple important details involved, such as showing that `i` remains an integer throughout this loop. We also note that the optimization is only valid if `arr.length` is in fact an integer value. If, for instance, `arr` was an object and `length` was a floating-point value, we could not safely remove the overflow check. Hence, eliminating these checks requires some kind of type analysis to prove the required invariants and is not a simple question of pattern matching on ASTs.

8.1.2 The Optimization

Overflow check elimination, in the simple form described just described, which is able to eliminate overflow checks on array index incrementations, was easily implemented as part of BBV. Two simple additions were needed, and just a handful of lines of code were changed.

The first necessary component was to enrich our notion of integer types with a one-bit flag which serves to indicate that an integer value is “submaximal”, that is, that the value is known to be less than the maximum value representable as an integer in our system. This flag is automatically set on the true branch of less-than integer comparisons.

The second component is the elimination of overflow checks using the newly enriched type information. This simply required modifying the code generation for integer additions not to insert an overflow check when one of the operands has the "submaximal" flag set and the other operand is the integer one.

This optimization was not implemented when the eager BBV paper [5] was submitted, and was disabled for the ECOOP 2014 lazy BBV article [7], so as to not misrepresent performance gains obtained when using BBV to perform type specialization. It was enabled for subsequent submissions.

8.1.3 Results

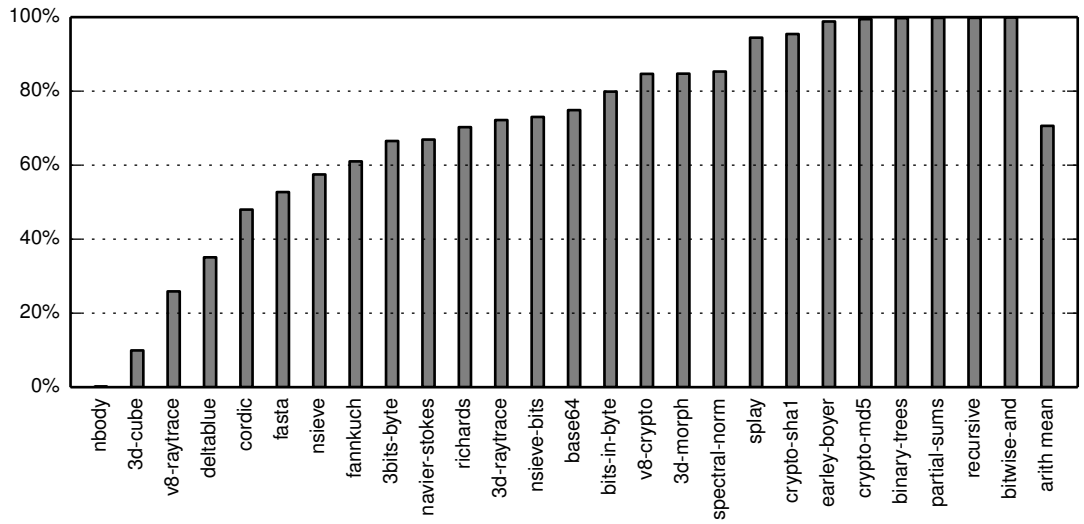

Figure 8.2: Number of overflow checks executed relative to a baseline without overflow check elimination

To evaluate this optimization, the same 26 benchmarks were used as in Chapter 7. The graph in Figure 8.2 shows the proportion of overflow checks remaining after the overflow check elimination was enabled compared to a baseline without this optimization. As can be seen, the optimization is very effective, eliminating most of the overflow checks in benchmarks making heavy use of the `for` loop iteration pattern relative to

other overflow-prone integer operations. On average, 29% of overflow checks are eliminated across our benchmarks.

We note that the `bitwise-and` microbenchmark, which contains a `for` loop, does not show any overflow checks eliminated. This is because this benchmark operates on global variables. We have made the technical decision of not versioning object shapes based on the "submaximal" flag.

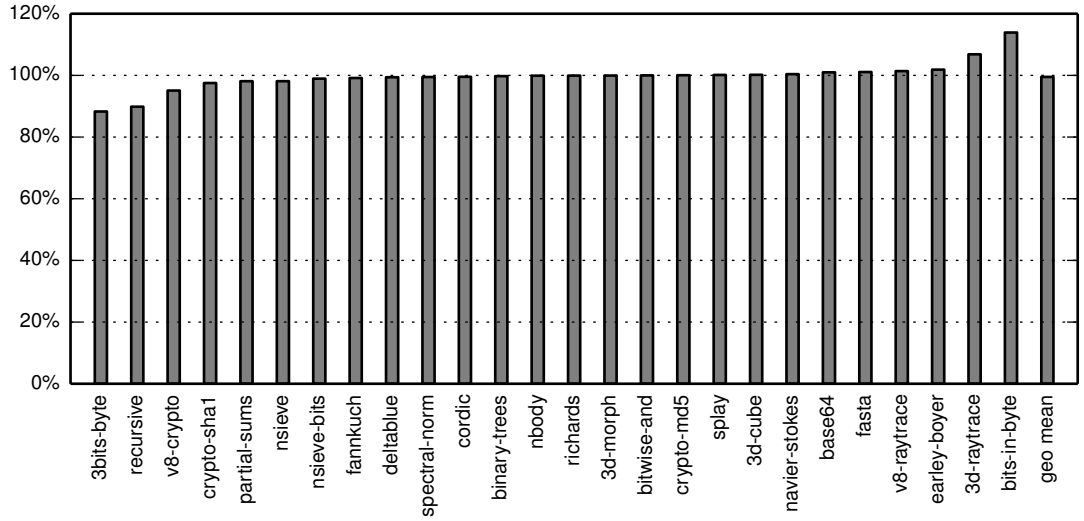

Figure 8.3: Execution time relative to a baseline without overflow check elimination

Enabling overflow check elimination produces a negligible reduction in machine code size, just 0.1% on average. Figure 8.3 shows the execution time after overflow check elimination is enabled compared to a baseline without it. Some benchmarks show speedups and others show a slowdown, but this appears to be due to machine code alignment effects.

The optimization, though it does effectively eliminate overflow checks, does not significantly affect performance. On most benchmarks, the execution time is unaffected. We believe this is because overflows almost never occur, and so overflow checks are easily predicted by modern x86 hardware. There is a possibility that such an optimization could be relevant on other platforms, or once more significant sources of overhead are optimized away.

Although this optimization is not a success story in terms of performance numbers, we believe it to be an interesting demonstration that BBV has other potential applications besides type check elimination.

8.2 Interprocedural Shape Change Tracking

8.2.1 Problem Description

In Chapter 7, we introduced a speculative optimization used to propagate return types from callees to callers without dynamic dispatch overhead. The technique essentially predicts future return types of a function and deoptimizes call continuations if this prediction later turns out to be incorrect. We have used a similar strategy in an attempt to improve the results of intraprocedural object shape propagation (see Chapter 6).

An issue with shape propagation which we have not discussed in much detail is that of aliasing. In JavaScript, objects are manipulated through references. As such, two distinct variables can refer to the same object because of pointer aliasing. This means that the shape of an object referenced by some variable `a` can change if we perform property writes on some other variable `b` that happens to refer to the same object. This places limitations on shape propagation, because a mutation to one object may cause known object shapes to become invalidated elsewhere.

In the intraprocedural case, if we know that two object references point to objects of different shapes, we can trivially infer that these two references do not alias [15], which is very convenient. Things are more complicated once function calls come into the picture, however. In general, we do not know which objects may be written to by a given callee. Furthermore, one function call may produce a deep nesting of subsequent calls, which may also modify objects. The easy solution is to stop shape propagation as soon as a function call is encountered, but this is inefficient.

8.2.2 The Optimization

We have implemented a speculative solution similar to the return type prediction from Chapter 7. A per-function flag serves to indicate whether or not a given function

may modify object shapes during its execution. Initially, each function is assumed not to modify any shapes, but if the function does modify shapes at execution times, this may result in the deoptimization of call continuations.

Call continuations rely on the shape modification flag to decide whether or not they should keep known shape information from before the call site. That is, when a callee may modify object shapes, known shape information at the call site is discarded.

This system is able to get around some of the precision limitations of static analyses. That is, with a static analysis, it may be difficult to guarantee that a given function will not call a chain of functions which eventually modify object shapes down the line. Our speculative system, however, takes run-time behavior into account. Note that no dynamic checks are involved. The JIT compiler sets the shape modification flag at code generation time, when object property write code is compiled.

8.2.3 Results

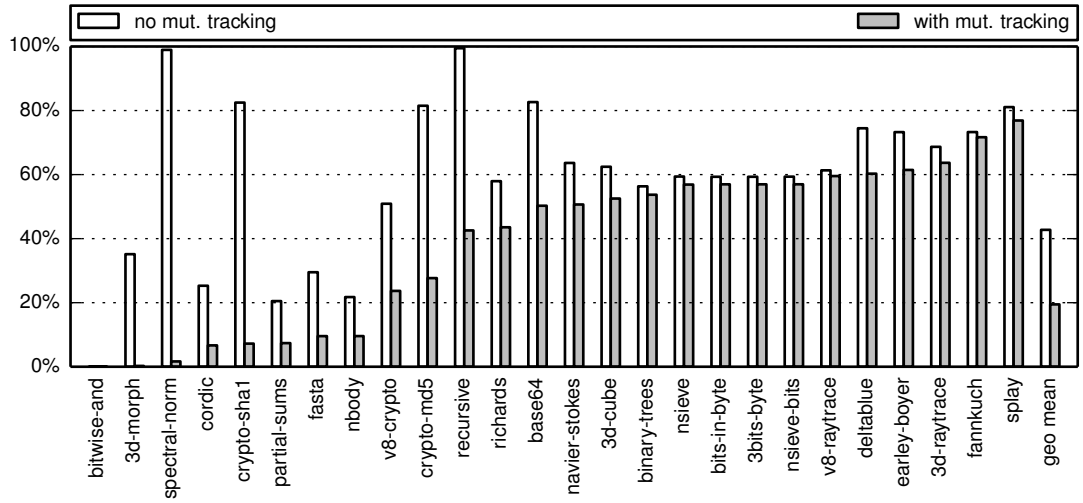

Figure 8.4: Shape tests relative to a baseline without shape propagation

Figure 8.4 shows the number of shape tests with and without interprocedural shape change tracking, both relative to a baseline without shape propagation. A reduction in the

number of shape tests is obtained on almost every benchmark. The results are impressive for some of the smaller benchmarks, but less so for more complex ones.

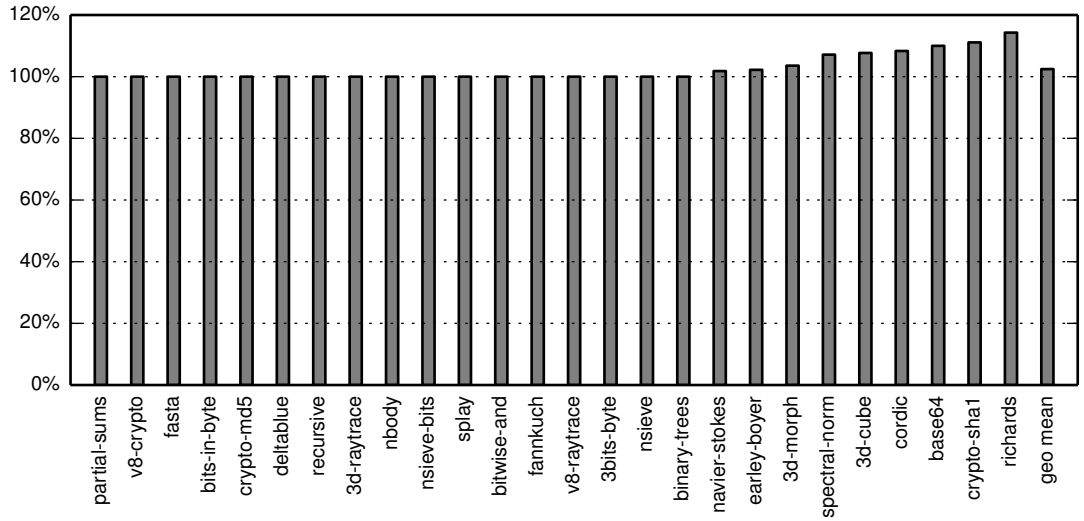

Figure 8.5: Execution time relative to a baseline without shape mutation tracking

In terms of execution time (see Figure 8.5), several benchmarks show a clear deterioration of performance. We suspect that this is because shape change tracking causes more invalidations, which result in deteriorated code quality. The numbers support this explanation. Enabling interprocedural shape change tracking multiplies the number of call continuations invalidated by a factor of six.

When a call continuations is invalidated, a new call continuation may later be recompiled. However, this new continuation is generated out of line. That is, recompiled call continuations break the linear flow of generated machine code, and may cause instruction cache misses or branch mispredictions. We believe this problem could be addressed through adaptive recompilation (see Section 9.3).

8.3 Versioning and Register Allocation

8.3.1 The Problem

Higgs uses a greedy algorithm to perform register allocation. Register allocation decisions are made on the fly during code generation. When an IR instruction is compiled to machine code, its operands are allocated registers. This may cause previously allocated values to be spilled onto the stack if no registers are currently free. The greedy register allocation algorithm is simpler and faster than more sophisticated algorithms such as graph coloring. It behaves best when the register pressure is small and control flow is fairly linear.

The main weakness of greedy register allocation is that it has a myopic view of the code. That is, it makes individual register allocation decisions with limited awareness of their global impact. One area where performance is often lost is at loop back edges. The greedy allocator allocates registers for the loop header block when a control flow edge first enters the loop. Later, a loop back edge jumps back to the loop header, but the mapping of live variables to registers and stack slots often does not match the allocation made when first entering the loop.

Transitioning from one register and stack slot assignment to another requires the introduction of move instructions along the back edge. In other words, several move instructions may be executed for each loop iteration, which is inefficient. In particular, moving values from registers to stack slots and vice versa is particularly costly.

8.3.2 The Optimization

We have enabled the versioning of basic blocks based on register allocation state. This was a very simple modification, requiring the addition of just two lines of code in the version difference scoring function (see Chapter 5). The two new lines of code are shown in Figure 8.6. These add a penalty score for each value which would be spilled (written to the stack) for a given branch edge. The net result of this addition is that new block versions will be created to avoid spills when possible.

Should a loop back edge have a given variable mapped to a register, and the loop

```
// If this would cause a spill, add a penalty
if (predSt.isReg && succSt.isStack)
    diff += 1;
```

Figure 8.6: Two new lines of code enable versioning based on allocation states

header have the same variable spilled on the stack, the BBV algorithm will try to create a new loop header block version where the variable is assigned to a register, so long as the block versioning budget is not exceeded. This scheme can effectively unroll one or more iterations of a loop so as to try and keep loop variables in registers instead of writing them to the stack.

8.3.3 Results

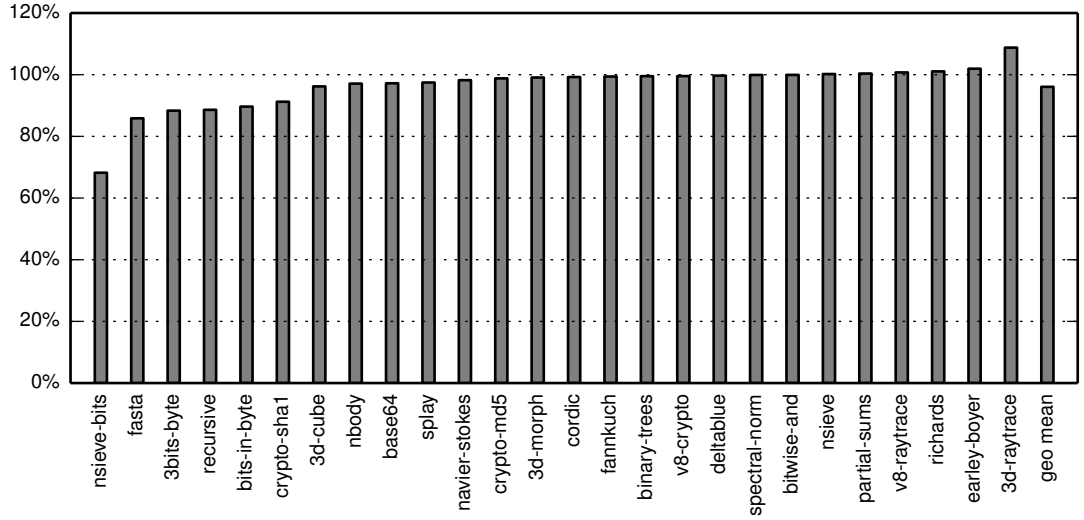

Figure 8.7: Execution time relative to a baseline without versioning based on register allocation state

Figure 8.7 shows the execution time with register-based versioning, compared to a baseline without it. The introduction of this optimization reduces execution time by 4% on average and 32% in the best case. Notably, the 9 benchmarks which benefit most from this optimization are all loop-intensive.

Register-based versioning causes a code size increase of 0.5% on average and 4.1% in the worst case.

8.4 BBV and the Instruction Cache

In Chapter 4, we have presented an implementation of BBV which generates block versions eagerly. This implementation successfully eliminates a large proportion of dynamic type tests, but also significantly increases code size, and fails to produce measurable performance improvements. We have speculated that the lack of performance improvements was due to an increase in instruction cache misses as a result of the increased code size. As of today, most modern CPUs have a relatively small 32KB instruction cache. Hence, it seems plausible that a code size increase may result in an increase in the number of instruction cache misses. However, when the CC 2015 and ECOOP 2015 articles were submitted, we had not validated this assumption.

We have set up a comparative experiment to examine the effect of laziness on the number of instruction cache misses. For this, we have used the perf Linux tool [1], which gives access to performance counters present in modern CPUs. A version of Higgs which can be made to generate versions either lazily or completely eagerly was used, as in the ECOOP 2015 article (see Chapter 5) with `maxvers=5`. The tests were performed on a computer equipped with a Core i7-4771 CPU and 16GB of RAM running Ubuntu Linux 14.04. The same 26 benchmarks were used as in the rest of this thesis. Benchmarks were given one warmup run, and then iterated until there were at least 60 seconds of execution time, so as to to offset compilation time.

With the aforementioned configuration, completely eager BBV yields a code size 416% larger than lazy BBV on average, and produces code which executes 119% slower than with lazy versioning. Note that much of the versions produced by eager BBV, although they are generated, are never actually executed. The graph shown in Figure 8.8 shows the relative number of instruction cache misses obtained with the eager configuration compared to lazy BBV. On average, eager BBV yields 254% more instruction cache

1. https://perf.wiki.kernel.org

misses. In every case, it increases the number of instruction cache misses, sometimes drastically.

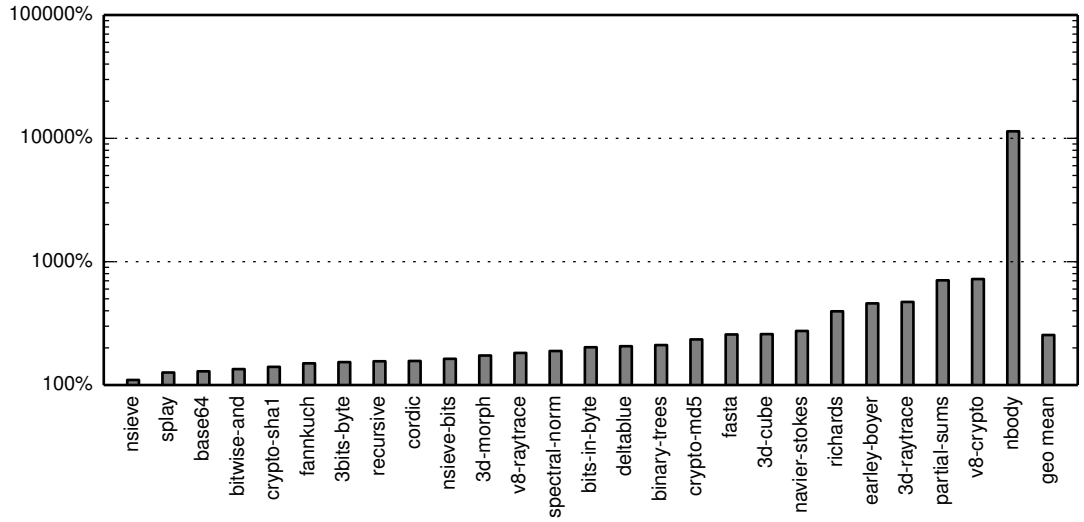

Figure 8.8: Instruction cache misses with eager BBV, relative to lazy BBV

Another advantage of the lazy approach is that it can arrange block versions in memory based on the order in which they are first executed. This tends to produce more linear machine code orderings with less jump instructions and less out of line jumps. As such, it is very likely that eager BBV produces more branch mispredictions than the lazy approach. We have taken to empirically validating this hypothesis as well.

Figure 8.9 shows the number of branch mispredictions measured with eager BBV relative to the number measured with lazy versioning. The impact of laziness is not as clear cut as with instruction cache misses. Some benchmarks, in fact, have less branch mispredictions with eager BBV. However, on average, the eager approach results in 128% more branch mispredictions.

8.5 Microbenchmarks

Figure 8.10 is a comparison of the performance of Higgs against that of V8 version 3.29.66 and SpiderMonkey version C40.0a1 on a few microbenchmarks written by us to

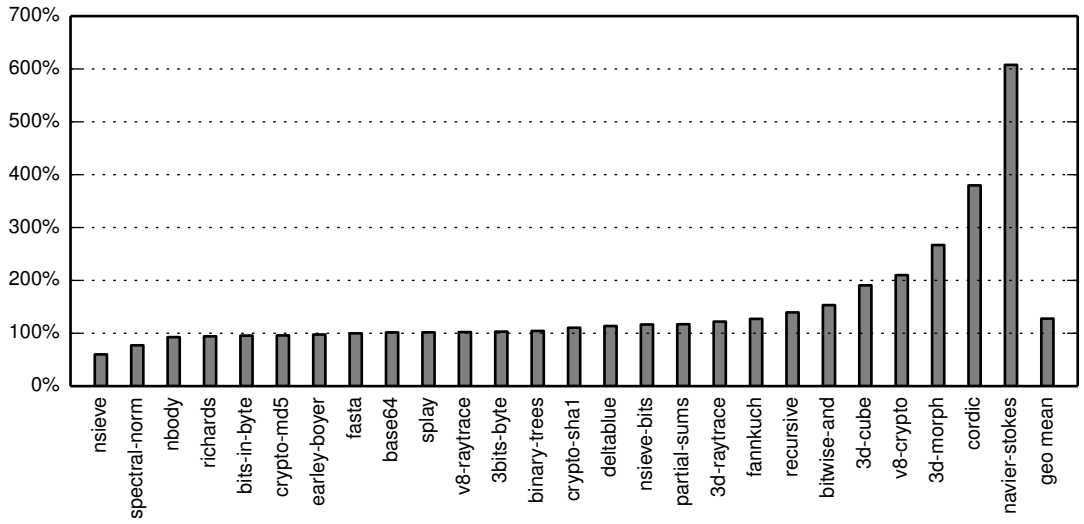

Figure 8.9: Branch mispredictions with eager BBV, relative to lazy BBV

examine the performance of specific language constructs. The source listings of these microbenchmarks can be found in Appendix II.

fib44. The `fib44` microbenchmark computes the 44th fibonacci number recursively. This benchmark showcases how Higgs is able to optimize global variable accesses and reflects the performance of the calling convention used by Higgs.

loop-global-incr. The `loop-global-incr` microbenchmark increments a global variable in a loop, similarly to what is done in `bitwise-and`, confirming that Higgs currently has faster global variable accesses than both V8 and SpiderMonkey.

list-sum. We wrote the `list-sum-100` microbenchmark to investigate the raw object property read performance of Higgs. It computes the sum of integer values stored in nodes of a linked list. On this benchmark, Higgs outperforms V8 by a narrow margin, but performs slightly worse than SpiderMonkey. However, if we increase the length of the linked list to 1000, as is done in `list-sum-1000`, we see that Higgs then performs worse than both V8 and SpiderMonkey, likely because they use smaller object layouts which fit better in the L1 CPU cache. This shows that when a program is memory bound, machine code quality is not the only relevant factor.

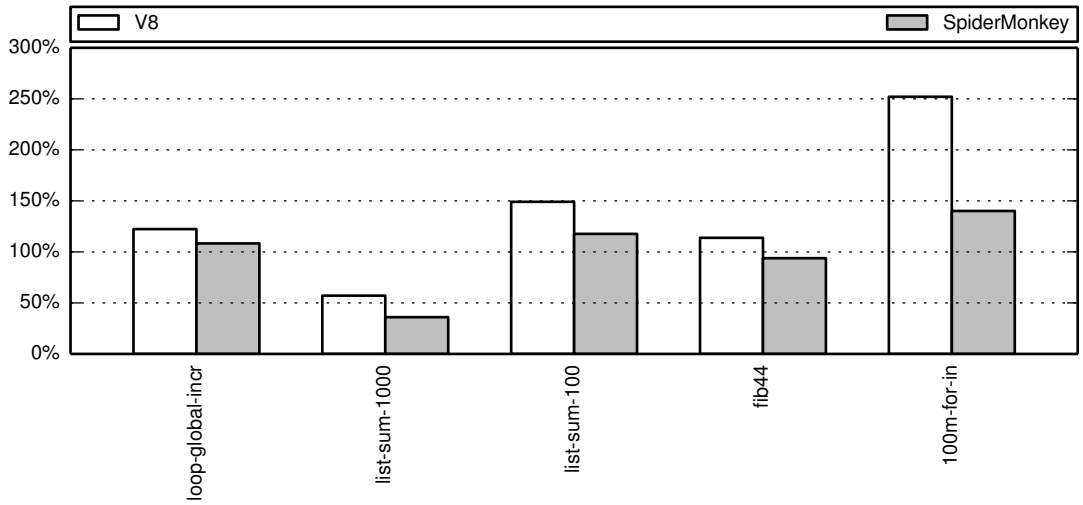

Figure 8.10: Speedup of Higgs over V8 and SpiderMonkey (higher favors Higgs)

100m-for-in. Finally, the `100m-for-in` microbenchmark showcases the performance of Higgs when iterating over object property names in a `for-in` loop. We have spent considerable time optimizing the performance of this specific language feature in order to reduce the performance disadvantage of Higgs on the `fasta` benchmark, and came to develop a more efficient implementation than both V8 and SpiderMonkey. This microbenchmark demonstrates the power of micro-optimizations. Perhaps V8 and SpiderMonkey outperform Higgs on average because they have had years to develop a wide range of micro-optimizations which Higgs does not yet possess.

8.6 Summary

This chapter presents a set of yet unpublished experiments with interesting results. Three of these investigate the potential of the BBV technique in areas not directly related to dynamic type check elimination.

Our first experiment explores the use of BBV to perform some basic overflow check elimination. Although no measurable speedups are obtained, we consider this experiment successful, as a large proportion of overflow checks are eliminated, with only

small changes required to the Higgs implementation to enable this.

The second experiment is an implementation of the interprocedural tracking of shape changes using a speculative deoptimization mechanism. The approach is successful in terms of eliminating dynamic shape tests. However, the performance is unfortunately slightly degraded. This is likely a limitation of the current implementation, which could be addressed using adaptive recompilation (see Section 9.3).

The last experiment in extending BBV explores versioning based on register allocation states. With only some trivial modifications to the Higgs source code, we are able to achieve large speedups on several benchmarks.

Also included in this chapter is an investigation of the effect of eager and lazy generation of block versions on instruction cache misses and branch mispredictions. Empirical results show that eagerly generating block versions leads to more instruction cache misses on every benchmark, and that the difference can be dramatic in some cases. This supports the hypothesis that instruction cache misses are the main reason why we were unable to obtain performance improvements with eager BBV.

Finally, we tested the performance of Higgs, V8 and SpiderMonkey on a few microbenchmarks of our own design. We showed that Higgs does outperform both JS engines in specific instances. This is also a demonstration that the overall performance of a JS VM depends on many orthogonal micro-optimizations.

CHAPTER 9

FUTURE WORK

Over the last few years, I have had time to think about potential avenues of future research. Several of these are enumerated in the following subsections, starting with those I believe to be the most promising. I believe that with some or all of the following enhancements, BBV could be made to shine even more.

9.1 Incremental Inlining

Inlining is one of the most important optimizations a compiler can perform. It is crucial for maximum performance, particularly in modern object-oriented or functional languages where many small functions are used. I strongly believe that an important direction to explore in future work would be the extension of BBV with incremental inlining. That is, the ability to inline callees one basic block at a time, as a natural extension to the versioning process.

Currently, Higgs uses simple heuristics to inline a fixed set of runtime primitives, but does not inline regular function calls. The inlining of runtime functions is very costly, both in terms of memory and time, as it requires the entire AST of callees to be copied and inserted into the caller. Because runtime primitives are generic and must deal with multiple input types, much of the inlined code ends up never being executed, making the entire process wasteful.

TraceMonkey is still very competitive with V8 on some benchmarks. Its power lies in its capacity for deep inlining. Augmenting BBV with incremental inlining could allow speculative and deep inlining of calls, making the technique much more competitive, both in terms of execution time and compilation time. Only the basic blocks inside a callee that are executed in the inlining context would be processed, meaning the inlining process itself would be faster.

9.2 Multi-Stage Fallback

The current implementation of Higgs has a maximum number of versions that may be generated for any given block, the `maxvers` parameter. When this limit is hit, a generic version is generated, with all type information eliminated. The generic version is guaranteed to be compatible with any incoming types.

Conceivably, instead of generating a generic version right away, a first generic version could be generated which only removes the type information that differs between previously requested versions, but keeps the types that remain the same. In most cases, it is likely that one or maybe two types are highly polymorphic and causing the version limit to be hit, but most of the known type information remains the same.

By generating a "compatible" but not fully generic version for fallback, much of the known type information could be preserved, allowing the system a form of graceful degradation. In the rare case where further incompatible versions are requested past the version limit, a truly generic version can be produced as before.

Multiple variations on this scheme are possible, such as the use of a gradual degradation scheme where the version limit is set to a low value, such as 3, and each new version requested removes information about the live value causing a mismatch.

9.3 Adaptive Recompilation

In general, the more specific the type information propagated, the more pressure is placed on the BBV scheme. Some values, such as object shapes, are sometimes disproportionately polymorphic. For instance, while most local variable types have only one or sometimes two possible type tags, object references can easily have 5 or 15 possible object shapes in some cases. These highly polymorphic values are usually the ones causing the version limit to be hit.

Paradoxically, type information about megamorphic values is probably not worth propagating. If a value can have a large number of different types, then trying to create many different versions to propagate all these types will, in all likelyhood, result in unnecessary instruction cache pressure and have us hit the version limit faster. The

goal of BBV is not to create many versions, but to propagate useful type information effectively.

We have already shown in Chapter 7 that it is easy to invalidate and recompile block versions by turning them back into stubs. I would argue that when the block limit is hit for a given version, or when a function has too many versions, the ideal strategy may be to throw away some or all of this code and lazily recompile it.

Throwing away block versions could allow us to "un-propagate" some type information. For instance, when a given value which is found to be megamorphic, it may not be worth propagating more than one type for this value. It might be best, in fact, not to propagate any type information at all in some cases. Such a strategy could allow BBV to better deal with pathological cases and to better scale to larger, more complex programs. A strategy for "un-propagating" information about megamorphic values also paves the way for the propagation of more precise information. For instance, we may wish to enable BBV to propagate numerical ranges or even constants.

9.4 Adaptive Ordering of Type Tag Tests

BBV exploits the type tag tests already present in runtime primitive functions to extract type information. When type information is already known as part of the versioning context, redundant type tests can be eliminated. One weakness of the current approach, however, is that type tests are executed in a specific sequence, which is fixed and specified by the source code of primitive functions.

Consider the `$rt_toString` runtime primitive function listed in Figure I.2 of Appendix I. The type tag of the input value `v` is tested in a fixed sequence of type tag test instructions. When the type of `v` is known, the code generation can eliminate all redundant clauses and skip directly to the correct one, producing efficient code.

The weakness of this approach, however, is that when the type of `v` is unknown, multiple type tags tests may be executed before the correct type tag is found. For instance, if a boolean value was passed as argument, all other possible type tags would be tested before the correct one is found. This is highly inefficient.

Runtime primitives often perform this kind of dispatching based on type tags. To improve the performance of such type-based dispatch, a smarter solution may be to implement a kind of `switch` or `dispatch` statement which can dispatch to the correct operations based on type tags. The key difference here, would be that unlike a cascade of switch statements, there would be no implicit ordering between the different clauses.

To generate code for such a statement, the JIT compiler could lazily generate type tag tests based on the values encountered during execution. This means that value tags which are never encountered would never be tested for. The main heuristic this idea relies on is that values which are more frequent are more likely to be encountered first.

9.5 Propagating Facts instead of Types

In Chapter 8, we explored a few ways in which BBV can be used for optimizations that go beyond eliminating type tests. In order to truly exploit the power of BBV, it may be desirable to generalize the BBV architecture. In particular, there is a need to represent more detailed information about values.

In the current system, each value has one possible type tag and one possible known shape. One can imagine propagating more precise type information, such as sets of possible type tags, numerical constants, numerical ranges, string constants, etc. Going even further, it may be interesting to think of associating values with arbitrary sets of known facts linked by logical conjunctions, disjunctions and negations.

A more flexible system using logical connectives would allow us to know that a specific value has two possible types, and also to propagate negative information, that is, that a given value does not have a given type. Certainly, propagating large tree structures representing type information about a value could easily become unwieldy, but if these trees were immutable, they could be shared, and compared for equality with a simple pointer comparison. I believe that a richer value representation such as this can be optimized for performance without much difficulty.

9.6 Array Specialization

Because JS code makes extensive use of objects, we have made considerable efforts to optimize object property accesses in Higgs, with great success. However, one obvious limitation of the current system is that it does not introspect array element types. Arrays are simply seen as black boxes containing values of unknown types. Optimizing array element accesses would enable BBV to eliminate even more type tests.

PyPy [3] and V8 [12] already implement strategies to optimize both the in-memory layout and the dynamic overhead of arrays containing only integers or only floating-point numbers, which is a common use case. They do this by implementing specialized storage strategies (backing stores) for these specific kinds of arrays.

BBV could conceivably be extended to discover and propagate information about array value types. In Chapter 6, we have shown how the notion of types used by BBV can be extended to accomodate for the concept of typed object shapes. A similar extension could be made to propagate information about array element types. This would make it possible to say that a given value is an array containing elements of one or multiple types, and specialize generated machine code in consequence.

9.7 Array Bounds Check Elimination

Section 8.1 explains how BBV can be extended to eliminate overflow checks associated with array index incrementations. This optimization propagates a one bit flag telling us that an integer index value can be incremented without overflow. The flag propagates from the loop test condition (usually of the form `i < arr.length`) to the index incrementation operation (usually of the form `i++`), allowing the elimination of overflow checks.

The representation of types used by BBV could be enriched to propagate more detailed facts (see Section 9.5) arising out of comparison operations. For instance, instead of propagating just one bit indicating that an integer index value is submaximal, we could directly propagate the fact that the index is non-negative and less than the length of some specific array. Propagating this information to array access operations would allow us to

eliminate array bounds checks.

There are some obvious complication here, in that that arrays are both mutable and subject to aliasing. As such, a fact stating that some index is within the bounds of an array can become invalid if the length of the array is reduced through other aliases. An easy solution to this problem is to simply discard such facts if any array has its length changed, or if any (non-inlined) function is called. A speculative deoptimization strategy such as that presented in Section 8.2 could potentially be used to speculate that callees will not shrink the length of any arrays.

9.8 Closure Variable Awareness

Another gap in the current system is that it does not introspect variables captured by closures (free variables). I believe that one possible approach to this problem may be to represent closures as objects, and captured variables as object properties. This would allow exploiting already existing typed shape mechanisms (see Chapter 6) to propagate information about captured variable types.

9.9 Allocation Sinking

Dynamic language implementations such as PyPy and LuaJIT perform an optimization known as allocation sinking [2]. This optimization is meant to avoid allocating temporary objects which are then freed shortly after. In tracing JITs, the optimization replaces an object allocation inside of a trace by a set of temporary variables representing its individual fields. These temporary variables can later be materialized into a real object allocation if control flow exits the trace and the object is still alive at the trace exit.

This optimization is particularly easy to implement in tracing JITs, but should be feasible to implement in a BBV system by essentially delaying object allocations, and indicating in the versioning context that the objects are stored in temporary values. The objects could then be materialized when some specific condition occurs, such as a return statement, or simply never allocated when this is possible.

I believe that allocation sinking would be particularly beneficial for functional code.

In such code, many temporary closure objects (and corresponding closure cells) are allocated. By combining allocation sinking with incremental inlining, it may be possible to optimize functional constructs such as `map` and `forEach` loops and make them as efficient as ordinary `for` loops.

9.10 Lazy, Deferred Computations

Some conference reviewers, upon seeing the early work on lazy BBV (see Chapter 5), have expressed doubts about its ability to compete with trace compilation. Traces are long linear segments of code which are easy to analyze and optimize in various ways. In contrast, BBV is a localized approach, which seems myopic, and fundamentally more limited in terms of optimization capabilities. For instance, it is easy to remove redundant computations from traces by moving them onto side-exits. How can BBV achieve the same thing, given that it knows nothing, when compiling a given block, about what computations will be happening in other basic blocks?

Partial allocation sinking (Section 9.9), is a technique that delays temporary allocations until we know they are really needed. This is an effective technique to eliminate unnecessary allocations. I believe that partial allocation sinking could be implemented in a BBV system by making values as deferred allocations in the versioning context.

The idea of deferred allocations could, in theory, be generalized to that of deferred computations. The idea would be to avoid performing a computation until we know its value is actually needed, or until we are forced by some boundary condition (e.g. side-effects) to perform the computation. The versioning context could represent values as trees of IR instructions to be evaluated. These trees would then be materialized into machine code when a return statement or a store to memory is encountered.

Deferred computations would make it possible for BBV to eliminate redundant computations, reorder instructions, and to perform higher-level optimizations such as optimizing successive object property reads. This approach may also enable better register allocation over larger sequences of instructions. One potential issue with this approach is that multiple paths through the control flow graph may end up performing the same

computation which could have been performed earlier, leading to code bloat. This problem, if it really is one in practice, could be alleviated by using adaptive recompilation (Section 9.3).

9.11 IR-level versioning

BBV, as presented in this thesis, is a technique that operates in a JIT compiler's backend and generates multiple versions of machine code corresponding to a given basic block at the IR level. Another possible approach would be to do versioning directly in the compiler IR. The compiler would then generate new type-annotated basic blocks in the IR itself.

One obvious advantage is that this would allow the compiler to perform higher-level transformations on the code. For instance, the current implementation of BBV cannot optimize successive object property reads effectively (e.g. `a = o.v.x; b = o.v.y`). In a system where IR nodes are allocated with object shapes, such an optimization becomes much simpler to implement.

Another advantage of IR-level versioning would be that by versioning at the IR level, it may be possible to represent versioning states more compactly than is currently done in the Higgs backend, by annotating the IR nodes.

9.12 Fragment Optimization

BBV interleaves compilation and execution. The technique makes use of stubs to detect which branch edges of a conditional branch are executed at run time. However, BBV is not limited to compiling one single basic block at a time. When encountering unconditional jumps, or branches whose direction can be determined at code generation time (e.g. type tests on variables whose type is known), BBV will compile multiple block in a linear sequence.

Since BBV can eliminate the large majority of type tests in a program (see Section 7), the compilation of long sequences of basic blocks is quite common in practice. This could be an opportunity to implement a mechanism to optimize multiple basic blocks

at a time. Multiple blocks chained together form what we might term a “fragment”, a sequence of instructions which must execute linearly, with a single exit at the end.

Instead of generating code for individual basic blocks directly in one pass, BBV could first do a type propagation pass with the purpose of determining how far a given fragment extends, that is, establishing how many consecutive branches are determined at code generation time. Then, a second pass could perform analyses and transformations at the IR level on the blocks which are part of the fragment.

Fragment optimization has obvious similarities with trace compilation, in that a fragment can be seen as a short trace without side exits. It has similar benefits as IR-level versioning (see 9.11), in that it allows optimizing higher-level constructs, such as consecutive object property accesses. Other benefits include the possibility of performing more sophisticated register allocation.

9.13 Garbage Collector Optimizations

It may be possible to optimize memory allocation and garbage collection using BBV. Systems which use a moving GC perform a check to verify if an object to be allocated will fit within the current heap. This check is typically performed at every allocation. With BBV, it may be possible to perform this check once for a small chunk of memory that can contain multiple objects. BBV can then be used to keep track, at code generation time, of how much space remains in the chunk. Essentially performing part of the allocation work at code generation time.

9.14 IR Performance Optimizations

Early on in the design of Tachyon and Higgs, we have made the choice to go for simplicity and maintainability in the implementation. Higgs is able to generate efficient machine code, but it was not optimized for fast compilation times. The data structures used to represent the IR and associated metadata are intended to be easy to manipulate and modify.

V8, in contrast, has been thoroughly optimized for fast compilation times. Whereas

Higgs uses the D GC to allocate ASTs and IR nodes, V8 uses a custom memory allocator called a "zone". These are flat arrays of bytes to which new objects are appended. One or more zones are associated with a given function, and all of the contents of one zone are deallocated at once when the function is no longer live.

In order to compete with V8 and SpiderMonkey on absolute compilation times, Higgs would need optimizations such as zones, as well as many other micro-optimizations.

9.15 Summary

In this chapter, I have outlined several potential improvements to the BBV technique which I deem to be interesting directions for future research.

There are, in my opinion, several "low-hanging fruits" which are very likely to pay off and require only small or moderate implementation effort. These are array specialization, closure variable awareness, multi-stage fallback and adaptive ordering of type tag tests. Array specialization and closure variable awareness address obvious limitations of the present work by enabling BBV to associate type information with language constructs which are not explicitly handled by the current system. Multi-stage fallback and adaptive ordering of type tag tests are promising improvements to the BBV technique which are almost guaranteed to produce better results in terms of type tag tests eliminated.

Incremental inlining, IR-level versioning and adaptive recompilation are significant time investments, but may allow BBV to get much closer to the performance levels of state of the art JS VMs. The current system needs a way to perform method inlining. Incremental inlining is an approach to this problem which meshes nicely with the BBV architecture, and is likely to reduce both execution and compilation times. IR-level versioning would allow a BBV compiler to perform high-level optimizations which are currently impractical to implement. Adaptive recompilation may allow BBV to propagate more precise type information while also guaranteeing that the approach scales to larger and more complex programs.

CHAPTER 10

CONCLUSION

This thesis is an initial exploration of BBV, an alternative JIT compilation strategy which aims to reach a good tradeoff between compiler architecture complexity and performance of the generated machine code. BBV uses the systematic versioning and specialization of individual basic blocks as a mechanism to accumulate and propagate contextual information. The technique optimizes code on the fly and in a single pass as it is generated, without the use of traditional program analysis techniques.

The first part of this thesis (Chapter 4) investigates an eager, intraprocedural form of BBV which attempts to generates all block versions for a given method at once, until a per-block version limit is reached. The approach is shown to be effective for eliminating redundant dynamic type checks which are part of the implicit semantics of dynamic programming languages. In our empirical tests, *eager BBV* is able to eliminate 64% of dynamic type checks on average, about twice as much as a fixed-point intraprocedural type propagation analysis. It also has much lower compilation time overhead. However, this strategy does not produce measurable performance improvements because eager versioning generates many block versions which are never actually executed, resulting in large machine code size increases.

Eager BBV generates more basic block versions than necessary because it is unable to effectively determine, at method compilation time, which versions are likely to be executed and which are not. As such, it must overapproximate the set of necessary block versions. Instead of relying on heuristics or trying to approximate this information using profiling techniques, we chose to explore the possibility of generating basic block versions lazily, on the fly (Chapter 5).

Lazy BBV uses machine code stubs to delay the generation of block versions until they are first executed. In this way, it is able to incrementally generate exactly the set of block versions necessary at run time. Using stubs and code patching, we implement this technique in a way that produces efficient linear sequences of machine code without un-

necessary jump instructions. By avoiding the generation of superfluous block versions, lazy BBV eliminates the code size bloat problem. Because it makes a better targeted use of the versioning budget, lazy BBV is also able to eliminate more type tests than the eager approach, 71% on average, and to deliver significant speedups compared to a system without BBV.

The basic BBV approach has no mechanism to attach property types to object properties. As a result, dynamic type tests are performed for almost every object property read. Since JS is an object-oriented language, and global variables are properties of the global object, the resulting overhead is significant. We devise a mechanism, which we refer to as *typed object shapes*, for attaching type information (including method identities) to object properties as part of the existing object shape metadata (Chapter 6). This mechanism is integrated with BBV by allowing the versioning scheme to capture and propagate object shapes. With typed shapes, the extended BBV system is able to eliminate more dynamic type checks than the plain intraprocedural approach, 79% on average, resulting in further performance improvements.

The intraprocedural BBV approach sees function calls as black boxes. That is, function parameters and return values are treated as variables of unknown type. This results in many redundant dynamic type checks being performed, particularly in recursive functions. We devise two separate mechanisms to propagate type information across function call boundaries (Chapter 7). *Entry point specialization* uses method identity information provided by typed shapes to pass argument types to specialized function entry points without dynamic dispatch. *Call continuation specialization* is a speculative strategy to pass return value types back to callers without dynamic overhead.

By combining lazy BBV, typed object shapes and the interprocedural extensions, we are able to eliminate 94.3% of dynamic type checks, on average, across our benchmark suite. We empirically show that BBV outperforms an idealized whole-program static type analysis with access to perfect information.

Finally, in Chapter 8, we show that BBV is usable for things other than type specialization. Because of its conceptual simplicity, BBV can be easily extended in various ways to incorporate new optimization capabilities. There are many possible avenues to

be explored (Chapter 9). Applications to other domains, extensions to make the technique more powerful as well as algorithmic refinements to improve the scalability of BBV while exploiting more opportunities for optimization. The work we present here is only scratching the surface of what is possible.

Compiler architecture is a very large problem space. There are many ways to design a compiler, each with unique tradeoffs. While commercial JS engines are unlikely to switch to BBV-based architectures overnight, it is our belief that code versioning with sub-method granularity is bound to eventually make its way into mainstream dynamic language VMs, in one form or another, as it is able to eliminate dynamic overhead untouched by traditional type analysis techniques.

BIBLIOGRAPHY

[1] Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb. LISP machine progress report. August 1977. URL http://dspace.mit.edu/handle/1721.1/5751.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation*, pages 43–52. ACM, 2011.

[3] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509531. URL http://doi.acm.org/10.1145/2509136.2509531.

[4] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74884. URL http://doi.acm.org/10.1145/74877.74884.

[5] Maxime Chevalier-Boisvert and Marc Feeley. Removing dynamic type tests with context-driven basic block versioning. *CoRR*, abs/1401.3041, 2014. URL http://arxiv.org/abs/1401.3041.

[6] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. *CoRR*, abs/1511.02956, 2015. URL http://arxiv.org/abs/1511.02956.

[7] Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 101–123, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.101. URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.101.

[8] Maxime Chevalier-Boisvert and Marc Feeley. Extending basic block versioning with typed object shapes. *CoRR*, abs/1507.02437, 2015. URL http://arxiv.org/abs/1507.02437.

[9] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 46–65, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_4. URL http://dx.doi.org/10.1007/978-3-642-11970-5_4.

[10] Maxime Chevalier-Boisvert, Erick Lavoie, Marc Feeley, and Bruno Dufour. Bootstrapping a self-hosted research virtual machine for JavaScript: An experience report. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4. doi: 10.1145/2047849.2047858. URL http://doi.acm.org/10.1145/2047849.2047858.

[11] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. ISSN 00029327. doi: 10.2307/2371045. URL http://dx.doi.org/10.2307/2371045.

[12] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2015, pages 105–117, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8.

doi: 10.1145/2754169.2754181. URL http://doi.acm.org/10.1145/2754169.2754181.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[14] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800542. URL http://doi.acm.org/10.1145/800017.800542.

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *SIGPLAN Notices*, 33(5):106–117, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277670. URL http://doi.acm.org/10.1145/277652.277670.

[16] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134780. URL http://doi.acm.org/10.1145/1134760.1134780.

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New

York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542528. URL http://doi.acm.org/10.1145/1542476.1542528.

[18] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, The University of Arizona. Department of Computer Science., Tucson, AZ 85721, USA, October 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.4394.

[19] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254094. URL http://doi.acm.org/10.1145/2254064.2254094.

[20] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178478. URL http://doi.acm.org/10.1145/178243.178478.

[21] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, July 1996. ISSN 0164-0925. doi: 10.1145/233561.233562. URL http://doi.acm.org/10.1145/233561.233562.

[22] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0. URL http://dl.acm.org/citation.cfm?id=646149.679193.

[23] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with

dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. URL http://doi.acm.org/10.1145/143095.143114.

[24] ECMA International. *ECMA-262: ECMAScript Language Specification*. European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edition, 2009.

[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

[26] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 320–339, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15768-8, 978-3-642-15768-4. URL http://dl.acm.org/citation.cfm?id=1882094.1882114.

[27] Alan C. Kay. The early history of Smalltalk. In *History of programming languages—II*, pages 511–598. ACM, 1996. ISBN 0-201-89502-1. URL http://portal.acm.org/citation.cfm?id=1057828.

[28] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064996. ACM ID: 1064996.

[29] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. URL http://llvm.org/pubs/2005-05-04-LattnerPHDThesis.html.

[30] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–, Palo Alto, California, Mar 2004. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[31] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part one. *Communications of the ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

[32] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[33] Baptiste Saleil and Marc Feeley. Type check removal using lazy interprocedural code versioning. In *Scheme and Functional Programming Workshop*, 2015.

[34] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. Technical Report CSL-79-11, Xerox Corporation, August 1979.

[35] David Ungar and Randall B Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22:227–242, December 1987. ISSN 0362-1340. doi: 10.1145/38807.38828. ACM ID: 38828.

[36] David Ungar, Craig Chambers, and Bay-wei Chang. Organizing programs without classes. *Lisp and Symbolic Computation*, 4:223–242, 1991. doi: 10.1.1.56.8715. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.8715.

[37] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference*

on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '14, pages 133–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2926-2. doi: 10.1145/2647508.2647517. URL http://doi.acm.org/10.1145/2647508.2647517.

[38] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587. URL http://doi.acm.org/10.1145/2384577.2384587.

[39] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581. URL http://doi.acm.org/10.1145/2509578.2509581.

Appendix I

Runtime Primitives

This appendix contains source listings for runtime primitive functions used in Higgs. These serve to illustrate the extended JS dialect used to implement the Higgs runtime library, and the way in which low-level type tag test instructions are used to perform dynamic dispatch based on value types.

The `$rt_lt` primitive, illustrated in Figure I.1, implements the JS less-than comparison operator (i.e. `a < b`). Low-level type tag test IR instructions are used to check if the arguments `x` and `y` are integers (`int32`), floating-point values (`float64`) or strings and take appropriate action. The integer case is the most common, and so it is handled first. Should both `x` and `y` be integer values, an IR instruction specifically implementing machine integer less-than comparisons is used. On the x86 platform, this translates into a `cmp` followed by a `jl` instruction.

The case where both inputs are floating-point values is similar, and uses machine instructions specifically for comparing double-precision floating-point values. If one value is `float64` and the other `int32`, then the latter is converted into a floating-point value, and a floating-point comparison is performed. Comparing two strings, or a number and a string, results in a lexicographical string comparison being used.

The `$rt_toString` primitive, shown in Figure I.2, is used internally by our JS runtime to produce a string representation for any JS value. This function would be called, for instance, if calling the `print` function (console printing) on a number, an array or an object. The primitive uses a long (and sometimes inefficient) cascade of dynamic type tag tests to choose the appropriate action for any input value type.

```
/**
JS less-than operator
*/
function $rt_lt(x, y)
{
    // If x is integer
    if ($ir_is_int32(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_i32(x, y);

        if ($ir_is_float64(y))
            return $ir_lt_f64($ir_i32_to_f64(x), y);
    }

    // If x is float
    else if ($ir_is_float64(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_f64(x, $ir_i32_to_f64(y));

        if ($ir_is_float64(y))
            return $ir_lt_f64(x, y);

        if ($ir_is_undef(y))
            return false;
    }

    var px = $rt_toPrim(x);
    var py = $rt_toPrim(y);

    // If px and py are both strings
    if ($ir_is_string(px) && $ir_is_string(py))
    {
        // Do a lexicographical comparison
        return $ir_eq_i32($rt_strcmp(px, py), -1);
    }

    return $rt_lt($rt_toNumber(x), $rt_toNumber(y));
}
```

Figure I.1: Primitive function implementing the JS less-than comparison operator

```
/**
Get the string representation of a value
*/
function $rt_toString(v)
{
    if ($rt_valIsObj(v))
    {
        var str = v.toString();

        if ($ir_is_string(str))
            return str;

        if ($rt_valIsObj(str))
            throw TypeError('toString␣produced␣non-primitive');

        return $rt_toString(str);
    }

    if ($ir_is_int32(v))
        return $rt_intToStr(v, 10);

    if ($ir_is_float64(v))
        return $rt_numToStr(v, 10);

    if ($ir_is_string(v))
        return v;

    if ($ir_is_rope(v))
        return $rt_ropeToStr(v);

    if ($ir_is_undef(v))
        return "undefined";

    if ($ir_is_null(v))
        return "null";

    if ($ir_is_bool(v))
        return $ir_eq_bool(v, true)? "true":"false";

    assert (false, "unhandled␣type␣in␣toString");
}
```

Figure I.2: Primitive function to convert JS values to strings

Appendix II

Microbenchmarks

This appendix contains source code listings for the microbenchmarks discussed in Section 8.5.

```
function fib(n)
{
    if (n < 2)
        return n;

    return fib(n-1) + fib(n-2);
}

fib(44);
```

Figure II.1: fib44

```
function test()
{
    // 2 billion iterations
    // Note: i is a global variable
    for (i = 0; i < 2000000000; ++i)
    {
    }
}

test();
```

Figure II.2: loop-global-incr

```
// Generate a linked list of length 100
var lst = null
for (var i = 0; i < 100; ++i)
    lst = { val: i, next: lst }

function listSum(lst)
{
    if (lst == null)
        return 0;
    else
        return lst.val + listSum(lst.next);
}

// 50 million iterations
for (var k = 0; k < 50000000; ++k)
    s = listSum(lst);
```

Figure II.3: list-sum-100

```
function test(o)
{
    for (var i = 0; i < 100000000; ++i)
    {
        // Note: k is a global variable
        for (k in o)
        {

        }
    }
}

var o = {
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: 5,
    f: 6
};

test(o);
```

Figure II.4: 100m-for-in