

**ESPECIALIZAÇÃO ESPECULATIVA DE
VALORES BASEADA EM PARÂMETROS**

IGOR RAFAEL DE ASSIS COSTA

**ESPECIALIZAÇÃO ESPECULATIVA DE
VALORES BASEADA EM PARÂMETROS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

Março de 2013

IGOR RAFAEL DE ASSIS COSTA

**PARAMETER-BASED SPECULATIVE VALUE
SPECIALIZATION**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

March 2013

© 2013, Igor Rafael de Assis Costa.
Todos os direitos reservados.

Costa, Igor Rafael de Assis

C837e Especialização especulativa de valores baseada em
parâmetros / Igor Rafael de Assis Costa. — Belo
Horizonte, 2013
xx, 65 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais. Departamento de Ciência da
Computação.

Orientador: Fernando Magno Quintão Pereira

1. Computação - Teses. 2. Linguagens de
Programação (Computadores) - Teses. 3. Compiladores
(Computadores) - Teses. I. Orientador. II. Título.

CDU 519.6*33 (043)



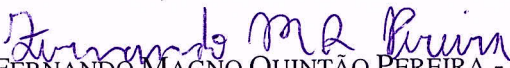
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

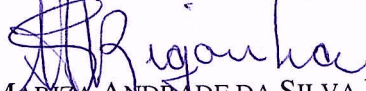
Especialização especulativa de valores baseada em parâmetros (Parameter-based speculative value specialization)


IGOR RAFAEL DE ASSIS COSTA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. FÁBIO MASCARENHAS DE QUEIROZ
Departamento de Ciência da Computação - UFRJ


PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG


PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 07 de março de 2013.

Acknowledgments

Ao meu orientador, Fernando Magno Quintão Pereira, por seus ensinamentos, críticas e sugestões que muito me ajudaram no desenvolvimento do meu trabalho. Eu tenho grande admiração pela sua dedicação e pelo seu apoio incondicional aos seus alunos. Espero trilhar o mesmo caminho quando trabalhar com meus alunos. Gostaria de agradecer também, ao Péricles e Henrique, que trabalharam comigo neste projeto com muito empenho e dedicação.

Aos meus pais, Geraldo e Rosa, e à minha irmã, Natália, pelo amor, apoio, incentivo e compreensão dedicados ao longo de toda minha jornada. À minha família, cujas reuniões sempre renovam minhas energias. À Júlia, pelo carinho e apoio incondicional, que desde o início do segundo tempo esteve sempre ao meu lado.

Aos amigos do LLP, Cristiano, Eliseu, Leo, João, Raphael, entre tantos outros, pelo excelente ambiente de trabalho. E também aos vizinhos do NPDI, pelos almoços e cafés mais que divertidos. Aos amigos da UFMG Informática Júnior, Pedro, Victor, Rafael, Felipe, Lucas, e tantos outros, pelo incentivo e inspiração em tantos projetos que resultaram nos caminhos que atualmente sigo. Aos demais amigos do DCC e da UFMG, cujas conversas e discussões valeram cada minuto: Guilherme, Débora, Cláudio, Thiago, Aline, Victor, Luciana, Rafael, Pedro, Fábio, Samuel, Raphaela, e tantos outros. E não menos importantes, os amigos de outros tempos: Wagner, Raphaela, Douglas, Giovanni, Rafael, Thiago, Daniel, Luiz, entre tantos outros.

Agradeço também aos órgãos de fomento, CNPq e CAPES, pela minha bolsa, a qual me propiciou dedicação integral à pesquisa.

Resumo

JavaScript emerge atualmente como uma das mais importantes linguagens de programação no desenvolvimento da interface com o usuário de aplicações Web. Desse modo, é essencial que os navegadores de Internet sejam capazes de executar programas JavaScript eficientemente. Entretanto, a natureza dinâmica dessa linguagem torna a eficiência de sua execução um desafio. Compiladores dinâmicos aparentam ser a ferramenta mais escolhida por desenvolvedores para lidar com tais desafios. Neste trabalho nós propomos uma técnica de especialização de valores especulativa baseada em parâmetros de função como uma estratégia para melhorar a qualidade do código produzido dinamicamente. Através de observação empírica, descobrimos que aproximadamente 60% das funções JavaScript encontradas nos 100 sites web mais populares são chamadas apenas uma vez, ou são chamadas sempre com os mesmos parâmetros. Baseado nessa observação, neste trabalho nós adaptamos diferentes otimizações clássicas de código para especializar código a partir dos valores atuais dos parâmetros de uma função. As técnicas propostas foram implementadas no IonMonkey, um compilador JIT para JavaScript de qualidade industrial desenvolvido pela fundação Mozilla. Por meio de experimentos executados em três coleções populares de testes, SunSpider, V8 e Kraken, foram alcançados ganhos de desempenho apesar da natureza especulativa da técnica proposta. Por exemplo, combinando algumas das diferentes otimizações propostas, obtivemos ganhos de 5.38% no tempo de execução na coleção SunSpider, além de reduzir o tamanho do código nativo produzido em 16.72%.

Abstract

JavaScript emerges today as one of the most important programming languages for the development of client-side web applications. Therefore, it is essential that browsers be able to execute JavaScript programs efficiently. However, the dynamic nature of this programming language makes it very challenging to achieve this much needed efficiency. The just-in-time compiler seems to be the current weapon of choice that developers use to face these challenges. In this work we propose Parameter-based Speculative Value Specialization as a way to improve the quality of the code produced by JIT engines. We have empirically observed that almost 60% of the JavaScript functions found in the world's 100 most popular websites are called only once, or are called with the same parameters. Capitalizing on this observation, we adapt several classic compiler optimizations to specialize code based on the run-time values of function's actual parameters. We have implemented the techniques proposed in this work in IonMonkey, an industrial quality JavaScript JIT compiler developed in the Mozilla Foundation. Our experiments, run across three popular JavaScript benchmarks, SunSpider, V8 and Kraken, show that, in spite of its highly speculative nature, our optimization pays for itself. As an example, we have been able to speedup the V8 benchmark by 4.83%, and to reduce the size of its native code by 18.84%.

List of Figures

2.1	How our work compares to the literature related to code specialization and speculation.	11
3.1	Histogram showing the percentage of JavaScript functions that are called n times	23
3.2	Histogram showing the percentage of JavaScript functions that are called with n different sets of arguments	24
3.3	Invocation histograms for three different benchmark suites	25
3.4	Invocation histograms for three different benchmark suites	26
3.5	The most common types of parameters used in benchmarks and in actual webpages.	28
3.6	The life cycle of a JavaScript program in the SpiderMonkey/IonMonkey execution environment.	29
3.7	(a) The JavaScript program that we will use as a running example. (b) The control flow graph of the function <code>map</code>	32
3.8	The result of our parameter specialization applied onto the program in Figure 3.7.	33
3.9	The result of applying constant propagation on the program in Figure 3.8.	34
3.10	The result of applying loop inversion on the program seen in Figure 3.9.	35
3.11	The result of applying dead-code elimination on the program seen in Figure 3.10.	36
3.12	The result of eliminating array bounds checks from the program shown in Figure 3.11.	37
3.13	The result of inlining the <code>inc</code> function in the code presented by Figure 3.12.	38
4.1	Size of generated code for SunSpider 1.0	48
4.2	Size of generated code for V8 version 6	49
4.3	Size of generated code for Kraken 1.1	50

List of Tables

3.1	Function Behavior Analysis Statistics	27
4.1	Run time speedup for the SunSpider benchmark	43
4.2	Run time speedup for the SunSpider benchmark (part 2)	44
4.3	Run time speedup for the V8 version 6 benchmark	45
4.4	Run time speedup for the V8 version 6 benchmark (part 2)	45
4.5	Run time speedup for the Kraken benchmark	46
4.6	Run time speedup for the Kraken benchmark (part2)	47
4.7	Code Size Reduction	47
4.8	Compilation overhead of different setups of our specialization engine	50
4.9	Percentage of additional recompilations	51
4.10	Number of deoptimizations	52

Contents

Acknowledgments	ix
Resumo	xi
Abstract	xiii
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Context	1
1.2 Contributions	2
1.3 Results	3
1.4 Outline	4
2 Related Work	5
2.1 JavaScript	5
2.2 Just-in-time compilation	7
2.2.1 Trace-based Just-in-Time Compilation	7
2.2.2 Method-based Just-in-Time Compilation	9
2.3 Code Specialization and Partial Evaluation	10
2.3.1 Control-flow Specialization	12
2.3.2 Data Specialization	13
2.3.3 Type Specialization	13
2.3.4 Value Range Specialization	14
2.3.5 Value Specialization	15
2.3.6 Input-centric compilation	18
2.4 Discussion	19

3	Parameter-based Value Specialization	21
3.1	Motivation	21
3.1.1	Methodology	21
3.1.2	Function call behavior	22
3.1.3	The types of the parameters	27
3.2	Parameter Based Speculative Value Specialization	28
3.2.1	The Anatomy of a MIR program	30
3.2.2	Parameter Specialization	31
3.3	Revisiting Classic Optimizations	31
3.3.1	Constant Propagation	31
3.3.2	Loop Inversion	35
3.3.3	Dead-Code Elimination	36
3.3.4	Array Bounds Check Elimination	37
3.3.5	Function Inlining	38
3.3.6	Other Optimizations	38
3.4	Discussion	39
4	Experiments	41
4.1	Benchmarks	41
4.2	Evaluation	42
4.2.1	Run time impact	42
4.2.2	Size of Generated Code	46
4.2.3	Compilation Overhead	48
4.2.4	Recompilations	49
4.2.5	Partial Specialization	51
4.2.6	Specialization policy	52
4.3	Discussion	53
5	Final Remarks	55
5.1	Future Work	55
5.2	Conclusion	55
	Bibliography	57

Chapter 1

Introduction

This dissertation is the result of two years of research on Just-in-Time compilation. Our findings are summarized in the five chapters that constitute this dissertation. In the first chapter we explain our motivations, goals and main contributions.

1.1 Context

Dynamic languages provide the ideal scenario for prototyping and developing new applications. In a world where new technologic startups are born every day, fast development is crucial to achieve its market as soon as possible. Rapid development cycles demand, from programming languages, features that allow ease of development, deployment, and portability which can be facilitated by high level of abstraction. Thereby, dynamic languages are being used to reduce development costs (Ousterhout 1998) and achieve flexibility in specific domains, like data processing (Meijer and Drayton 2005). JavaScript, Python and Ruby are successful examples of dynamic languages. They achieved high expressiveness and low learning curve (Gal et al. 2009) in a world of uncertainties and instability of software features.

JavaScript is presently the most important programming language used in client-side development of web applications (Godwin-Jones 2010). If in the past only very simple programs would be written in this language, like simple scripts used for creating menus on a web page or form validation, today the reality is different. JavaScript is ubiquitously employed into complex applications that consist of many thousands of lines of code. It is used to build applications as complex as Google's Gmail and Facebook, which are used by millions of users. Other applications, notably in the domain of image processing and gaming, are also becoming more commonplace due to the ease of software distribution. Furthermore, JavaScript, in addition to being

used directly by developers, is the target intermediate representation of frameworks such as Google Web Toolkit. Thus, it fills the role of an assembly language of the Internet. Given that every browser of notice has a way to run JavaScript programs, it is not surprising that industry and academia put considerable effort in the creation of efficient execution environments for this programming language.

However, executing JavaScript programs efficiently is not an easy task because it is a very dynamic programming language. JavaScript is dynamically typed: the types of variables and expressions may vary at run-time, forcing the compiler to emit generic code that can handle all potential type combinations. It also provides an `eval` function that can load and run strings as code. JavaScript programs tend to use the heap heavily, so the efficient management of the memory allocation is crucial to avoid performance degradation over time. This dynamic nature makes it very difficult for a static compiler to predict how a JavaScript program will behave at run-time. In addition to these difficulties, JavaScript programs are usually distributed in source code format, to ensure portability across different computer architectures. Thus, compilation time generally has an impact on the user experience. Today, the just-in-time compiler seems to be the tool of choice of engineers to face all these challenges.

A just-in-time compiler either compiles a JavaScript function immediately before it is invoked, as Google's V8 does, or while it is being interpreted, as Mozilla's TraceMonkey did. The advent of the so called *Browser War* between main software companies has boosted significantly the quality of these just-in-time compilers. In recent years we have seen the deployment of very efficient trace compilers (Chang et al. 2009, Gal et al. 2009, Mehrara and Mahlke 2011) and type specializers for JavaScript (Hackett and Guo 2012). New optimizations have been proposed to speedup JavaScript programs (Guo and Palsberg 2011, Sol et al. 2011), and old techniques (Chambers and Ungar 1989) have been reused in state-of-the-art browsers such as Google Chrome. Nevertheless, we believe that the landscape of current just-in-time techniques still offers room for improvement, and our opinion is that much can be done to improve run-time value specialization.

1.2 Contributions

We have observed empirically that almost 60% of all the JavaScript functions in popular websites are either called only once, or are always called with the same parameters, as we show in Section 3.1. Similar numbers can be extended to typical benchmarks, such as V8 (Google Inc. 2012), SunSpider (Apple Inc. 2012a) and Kraken (Mozilla

Foundation 2012b). Grounded by this observation, in this work we propose to use the run-time values of the actual parameters of a function to specialize the code that we generate for it. In Section 3.3 we revisit a small collection of classic compiler optimizations under the light of the approach that this work proposes. As we show in the rest of this work, some of these optimizations, such as constant propagation and dead-code elimination, perform very well once the values of the parameters are known. This knowledge is an asset that no static compiler can use, and, to the best of our knowledge, no just-in-time compiler currently uses.

Summarizing the contributions of this work we have:

- Behavioral analysis for JavaScript functions of the 100 most popular websites: we analyze the characteristics of function calls and their parameters, including their types;
- Open-source code for all optimizations implemented on IonMonkey (Mozilla Foundation 2012e): all the code developed is available in a public repository on the web <http://code.google.com/p/jit-value-specialization>;
- Evaluation of the Parameter-based Speculative Value Specialization approach in an industrial compiler of a dynamic language: we analyze the impact on run-time, code size and compilation overhead for widely used benchmarks.

The results of this research generated the following publications:

- Parameter Based Constant Propagation (SBLP 2012): this paper was a preliminary study of the technique that was only able to support artificial benchmarks (Alves et al. 2012);
- Just-in-Time Value Specialization (CGO 2013): this paper covers most of the work, pushing the implementation and evaluation far beyond the previous publication (Costa et al. 2013).

1.3 Results

We have implemented the ideas discussed in this work in IonMonkey, a JavaScript JIT compiler that runs on top of the SpiderMonkey interpreter used in the Firefox browser. As we explain in Chapter 4, we have tested our implementation on three popular JavaScript benchmarks: V8, SunSpider and Kraken. We evaluate two policies to parameter specialization: to specialize a function only one time and to specialize

a function two times. In the first, we only specialize functions that are called with at most one different parameter set. If a function that we have specialized is invoked more than once with different parameters, then we discard its binaries, and fall back into IonMonkey’s traditional compilation mode. In the second approach, we give one more chance to specialize a function, when some of its parameters remains the same. Even though we might have to recompile a function, our experiments in the SunSpider benchmark suite show that our approach pays for itself. We speedup SunSpider 1.0 by 2.73%. In some cases, as in SunSpider’s `access-nsieve.js`, we have been able to achieve a speedup of 38%. We have improved run times in other benchmarks as well: we have observed a 4.8% speedup in V8 version 6, and 1.25% in Kraken 1.1. This approach also achieves code size reduction by 16.72%, 18.84% and 15.94%, respectively in SunSpider, V8 and Kraken. For real web applications, like `www.google.com` and `www.facebook.com` we obtained 11.04% and 13.3% of code size reduction, respectively. Also the compilation overhead is almost zero, except for a few tests where the compilation time was increased by 3% and one case by 32%. We emphasize that we are comparing our research quality implementation with Mozilla’s industrial quality implementation of IonMonkey.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes important concepts related to this work and contextualize the Speculative Value Specialization in the Code Specialization scenario. Chapter 3 gives the motivation behind this work through behavioral analysis of real world JavaScript programs. It also explains how the parameter specialization can be done in the JIT compiler and how classic compiler optimizations can benefit from this specialization. Chapter 4 shows the experiments results that validate the feasibility of our approach, considering not only run time speedups but different forms of overhead. Chapter 4 validates, by means of an empirical evaluation, the feasibility of our approach. In that chapter we analyze the runtime, the size of the code that we generate and the compilation overhead that our techniques impose. Finally, Chapter 5 presents a conclusion and some discussion about this work, as well as an outlook on future work.

Chapter 2

Related Work

In this chapter we present a short introduction about the central topics related to this dissertation, which includes Dynamic Languages, Just-in-time(JIT) Compilation and Code Specialization. We start by explaining why some of the characteristics of JavaScript pose a challenge to efficient code generation. Then we present an overview and a comparison of two different JIT approaches: method-based JIT and trace-based JIT. We close this chapter by providing a taxonomy of different code specialization techniques. This taxonomy lets us explain precisely how our contribution is related to previous literature.

2.1 JavaScript

JavaScript is usually defined as an object-oriented language. However it employs a different object system based on the SELF language (Chambers and Ungar 1989), named prototype-based object system. This model does not define traditional object-oriented classes. Instead, it defines each object as a set of properties, which are a mutable map from strings to values. It uses prototypes instead of classes for implementing object inheritance. Each object has a special property that links objects through a chain, named *prototype*. This property can hold a reference to another object whose behavior is embedded into the current object. This means that each property lookup can result in searching not only the current object, but each parent linked through the prototype chain until the property is found. This model is very flexible but also the source of many optimization constraints.

One of these constraints is about the behavior of an object that can be modified dynamically through the change of its prototype property. In JavaScript, all functions are also objects which contains a prototype property referencing an empty object.

These functions can be used as constructors that clone the actual object into another object through the use of the keyword *new*. Naively, we can consider that each object has a unique shape, which restricts the possibilities of sharing an object shape to other similar objects. Some modern JavaScript engines may assume that an object shape remains stable through the program execution, and this assumption enables some JIT optimizations when creating new objects of a similar shape. However, as pointed by Richards et al. (2010), an object prototype hierarchy may not be invariant throughout the program execution. This emphasizes the need for a flexible and adaptive type inferencer, like the one proposed by Hackett and Guo (2012).

Some previous work have tried to use type inference as a way to generate fast code for dynamically-typed languages, such as JavaScript. Historically, these languages have run much slower than traditional statically-typed programming languages. For instance, the type of variables and expressions may change at run-time. Compilers for statically typed languages rely on stable type information to generate efficient machine code. However, this is not the reality for compilers that face dynamic typing. When exact type information is not available, the compiler needs to generate generalized machine code that can deal with all potential type combinations. This generalization often results in code that is slower than the code that a compiler could produce to a statically typed programming language. Therefore, type inference algorithms have a high impact over the performance of these languages. Although some of the type inference techniques seen in the compilation of statically typed languages compilation can be applied in the dynamic scenario, they imply in a prohibitive latency for a highly interactive user sessions in a web page. Recently, JavaScript was subject to intense research for type inference improvements (Anderson et al. 2005, Hackett and Guo 2012, Jensen et al. 2009, Thiemann 2005).

JavaScript, like other dynamic languages, was traditionally interpreted. Well-known examples of JavaScript interpreters are: Mozilla SpiderMonkey (Mozilla Foundation 2012d) and Rhino (Mozilla Foundation 2012c), Apple/Webkit SquirrelFish (Garen 2008), Microsoft JScript (Microsoft Corp. 2009). Mozilla developed two different engines for JavaScript: SpiderMonkey, implemented in C++, and Rhino, implemented in Java. Also, SpiderMonkey was the first JavaScript engine developed, written in 10 days by Brendan Eich (Eich 2011). They were all born as pure interpreters, at a time when webpages were much simpler than today. However, in the last years, the industry is putting much effort to develop efficient Just-in-time compilers to break the limitations of interpretation. Today there are many successful JavaScript JIT engines available, including Mozilla's JaegerMonkey and IonMonkey, Google's V8, Apple's Nitro and Microsoft's Chakra (Niyogi 2010).

2.2 Just-in-time compilation

Just-in-time compilers are part of the programming language's folklore since the early 60's. Influenced by the towering work of McCarthy (1960), the father of Lisp, a multitude of JIT compilers have been designed and implemented. These compilers have been fundamental to the success of languages such as Smalltalk (Deutsch and Schiffman 1984), Self (Chambers and Ungar 1989), Python (Rigo 2004), Java (Ishizaki et al. 1999), and many others. Part of this success is related with its benefits over traditional compilers, since there are more information available at run-time about a program. Some of this information can be very expensive or even impossible to obtain via traditional static analysis. In this section we will focus on the strategies that JIT compilers use to specialize code at run-time. For a comprehensive survey on just-in-time compilation, we recommend the work of Aycock (2003).

2.2.1 Trace-based Just-in-Time Compilation

The idea behind running specialized code sequences (traces) for frequently executed (hot) code regions seems to have first appeared in the Dynamo binary rewriting system (Bala et al. 2000). It aims to improve the performance of a instruction stream as it executes on a processor. Dynamo relies on run-time information to find the hot paths of a program and optimize its machine code. It was built to compile only single traces, despite linking them whenever possible. Dynamo was designed to be used both for dynamically generated streams and for the streams generated from the execution of a statically compiled native binary. It introduced native code tracing to perform profile-guided optimizations online, where the profile target was only the current execution. It also introduced the use of loop headers as candidates for hot traces. Dynamo has been a reference for almost all trace compilers that came after it.

Since the rise of Dynamo, many trace-based JIT compilers have been developed, like HotpathVM (Gal et al. 2006), YETI (Zaleski et al. 2007) and Maxpath (Bebenita et al. 2010b) for Java, PyPy (Bolz et al. 2009) for Python, SPUR (Bebenita et al. 2010a) and TraceMonkey (Gal et al. 2009) for JavaScript, LuaJIT (Yermolovich et al. 2009) for Lua, and Tamarin-Tracing (Chang et al. 2009) for ActionScript. All these trace compilers rely on the same core idea. They identify hot code regions, usually straight line sequences inside the target program. Yet, entire methods, or parts of a method can also be compiled. This is the approach that YETI adopts. Maxpath is an example of trace-based JIT that was designed to work without an interpreter. In this case, it uses a fast and non-optimizing baseline compiler to generate instrumented code that will

trigger the compilation of a hot trace. Trace-based compilers historically have treated mostly native code and typed languages, like Java, and therefore have focused less on type specialization and more on other optimizations. For dynamic typed languages, like JavaScript and Python, type specialization plays an important role in the optimization of the generated traces.

TraceMonkey (Mozilla Foundation 2012a) was the first JIT built by Mozilla to improve the JavaScript performance on the Firefox web browser. It was a trace-based JavaScript engine, developed to remove some of the inefficiencies associated with the interpretation of dynamic typing (Gal et al. 2009). TraceMonkey works at the granularity of individual loops, thus suiting well the needs of computing intensive web applications. It uses a mixed-mode execution approach: all programs are initially interpreted, and as the program runs, a profiler identifies frequently executed (hot) bytecode sequences. This hot bytecode is recorded, optimized and then compiled to native code. This native code will be executed for the next time the control-flow of the program reaches the loop header. As long as these sequences, called traces, remain type-stable, execution remains in the type-specialized machine code. This stability is controlled through a specific value-type mapping for each trace, which will be checked at run-time in a speculative approach.

This speculative nature of trace compilation implies the need to insert run-time checks into a trace code, in order to assert that all assumptions made during the compilation still hold. In many trace-based compilers, like TraceMonkey, these checks are called guard instructions. Each guard instruction consists of a group of instructions that performs a test and a conditional exit of the trace. These tests trigger events if there are changes in the expected execution flow. A typical change is caused by an else block that is taken after a few iterations in which the ‘then’ part of the conditional was taken. Other events that trigger side exits are changes in the types inferred dynamically. Each trace exit leads to a small piece of code called side exit. The code in these side exits restore the interpreter’s state. A side exit may also lead to the compilation of another trace, that follows an unvisited code path inside the hot loop. According to Gal et al. (2009), those side exits are heavily biased to not be taken, when types do not vary and a single control-flow is dominant.

However, guard instructions do not contribute to the computations of the original program and can be a significant fraction of total executed instructions. According to Mehrara and Mahlke (2011), the overhead caused by the execution of code only related to guards range from 22% to 42% of the total executed instructions, when analyzing well known benchmarks such as SunSpider and V8. Mehrara and Mahlke propose an execution strategy to reduce this overhead using a multi-threaded dynamically decou-

pled execution framework called ParaGuard. The authors claim that, across the 39 benchmarks studied, the ParaGuard technique achieves an average of 15% speedup over the original tracing technique. This speculative approach is more aggressive than traditional parallelization proposals like Ha et al. (2009), which presents a concurrent trace-based JIT that performs the compilation from LIR to native code as a background thread. The speedup achieved by this technique has average of 6% and a maximum of 25% speedup on the SunSpider benchmark suite. Although these two approaches are different ways to parallelize the execution, they are orthogonal and can be applied simultaneously.

Despite all recent effort to improve the performance of TraceMonkey, its development is now discontinued, and Mozilla’s effort has shifted to the improvement of JaegerMonkey (Mozilla Foundation 2012f) and subsequently to the development of IonMonkey. TraceMonkey is no longer part of Mozilla’s Firefox code since Firefox 11 (Nethercote 2012).

2.2.2 Method-based Just-in-Time Compilation

Method-based Just-in-Time Compilation works using methods and functions as the unit of compilation, instead of using only hot paths of the control-flow graph. This is considered the traditional approach to JIT compilation, not only for dynamic languages but for static ones too. This approach has some benefits over Trace-based compilation, like a possible larger optimization window, which is in this case the whole method. Some proposals, like Inoue et al. (2011), have tried to overcome this limitation. The work of Inoue et al. can span the scope of some methods through the use of nested trace trees. Even so, recursive function calls can be a hard issue to solve. Another advantage, when using method-based approaches within interpreters, is the reduction in the number of synchronizations points. They appear only at method call boundaries. However, method-based JITs have higher memory usage during compilation and optimization steps. They may also compile cold code regions that will never be executed. Therefore, trace compilation may be more suitable for resource-constrained devices, such as embedded environments.

Hybrid approaches that combines the best of method-based and trace-based JIT have also been evaluated for Java (Inoue et al. 2011). The authors propose a shared infrastructure between these two compilers: libraries, optimization passes, code generators, garbage collector. However, some optimizations can not be shared. For instance, Guo and Palsberg (2011) proved that some backward data-flow optimizations such as dead store elimination are not sound on trace compilation. Inoue et al. implemented

their trace-based JIT over the IBM J9/TR JVM and method-JIT, and despite being very limited (it is not integrated with a profiler) it generates code as fast as the baseline method-based compiler, only 8% slower on average. The Dalvik VM, the Android's Virtual Machine, was also proposed to be a hybrid compiler (Cheng and Buzbee 2010), using a Trace-JIT when running on battery and a Method-JIT in background while charging.

2.3 Code Specialization and Partial Evaluation

Specialization refers to the translation of a general function into a more limited version of it. For instance, a two-argument function can be transformed into a one-argument function by fixing one of its inputs to a particular value. However, if we consider a program text, rather than mathematical functions, we have what is called *program specialization* or *partial evaluation* (Jones et al. 1993). This process is performed by an algorithm usually called *partial evaluator*. Given a program and some information about its input values, a residual or specialized program is generated. If the specialized program is executed on the remaining input it will produce the same result of the execution of the original program on the complete input.

Code Specialization can be classified in various dimensions, regarding aspects like when it is applied or which part of code is affected. However, there is not a uniform taxonomy that is adopted among different research groups. For instance, code specialization can be done either at compile-time or run-time (Consel and Noël 1996), although this classification can appear with the names static and dynamic (code) specialization, respectively, in more recent works (Grant et al. 2000, Shankar et al. 2005, Suganuma et al. 2005, Zhang et al. 2007). In some works, like Canal et al. (2004), the term Value Specialization appears as an alias to Profile-guided Value Specialization. Value Specialization itself also appears with the name Value-based Code Specialization (Calder et al. 1999). As far as we now, there are no unified proposal to a more consistent and systematic classification system to code specialization.

Figure 2.1 is our attempt to bring some order to the code specialization literature. We have compiled this figure from descriptions found in several papers. Notably, we borrow Duesterwald (2005)'s suggestion to separate control-flow from data specialization. Although the present work only deals with speculative value specialization without a profiler, in the next sections we describe the other approaches listed in the tree. In this way, we hope to give the reader a good understanding on how this work compares to the existing literature. For instance, Duesterwald classifies code

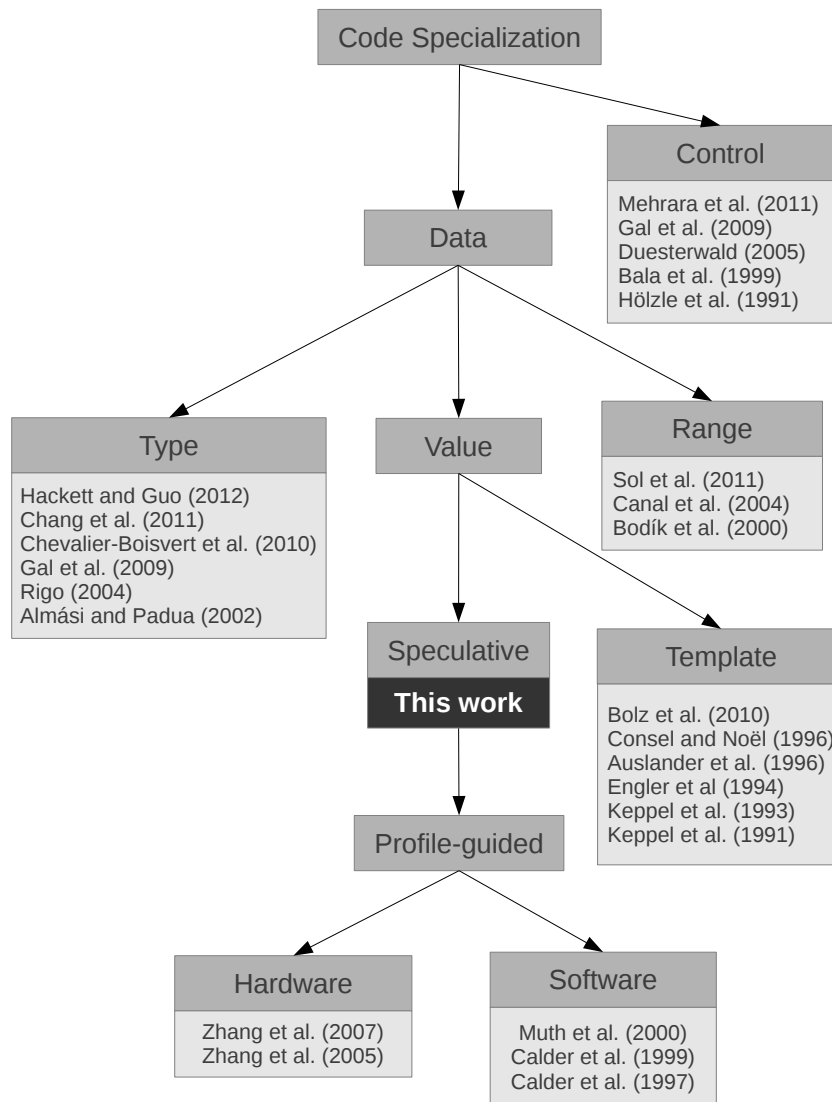


Figure 2.1: How our work compares to the literature related to code specialization and speculation.

specialization techniques in two main categories: control-flow specialization and data specialization. In the control-flow specialization group, we have optimizations that rely on the assumption that some paths in the program code will be traversed more often than others. On the other hand, data specialization is based on particular properties that hold for a certain data. Code specialization can also be done using a speculative approach. In this case, the compiler assumes that a particular property holds for a certain code or data and specializes it for that property.

2.3.1 Control-flow Specialization

Control-flow Specialization happens whenever the compiler assumes that a path will be often taken, and generates better code for that path. According to Duesterwald (2005), control specialization can be implicitly supported by trace selection, which can produce a new code layout, in the cache, specialized to the actual execution behavior. Partial procedure inlining (Duesterwald 2005), indirect branch linking (Duesterwald 2005), polymorphic inline caches (Hölzle et al. 1991) and code sinking (Bala et al. 1999) are all examples of Control-flow Specialization.

TraceMonkey was built on top of two different specialization techniques: control-flow specialization and type specialization (Gal et al. 2009). On the control-flow specialization side, the code is incrementally generated from the lazily discovered alternative paths of nested loops. TraceMonkey speculatively chooses a main path for a hot loop, and compiles it. All alternative paths are guarded and if one is taken the trace execution will stop, returning the execution to the SpiderMonkey interpreter through a side exit synchronization code. Whenever a side exit becomes hot, the trace compiler is triggered to compile the new trace for this branch. In the next executions, this trace will be called instead of returning control to the interpreter.

2.3.1.1 Speculative Control-flow Specialization

An example of control speculation, more specifically an example of Speculative Control-flow Specialization, has been recently proposed by Mehrara and Mahlke (2011) in the context of trace compilers. The authors propose that the main flow of a program's trace is executed by the main thread speculatively ahead, delegating to a second thread the task of executing the run-time checks. Whenever such event is detected, the two threads synchronize, the state of the interpreter is recovered, and execution falls back into interpretation mode. During the speculative execution, the execution of the main thread is sandboxed to make sure that no execution failures happen until all checks have been validated. The increase of performance are more evident in multicore systems with under-utilized cores, where the main and guard threads can execute concurrently. The control-flow is specialized in the sense that all side exits associated with the run-time checks are removed from the main execution flow.

Code sinking is also a Speculative Control-flow Specialization optimization. According to Bala et al. (1999), it is the symmetric counterpart to code hoisting for partial redundancy elimination. Its objective is to move instructions from the main execution path into fragment exits to reduce the number of instructions executed on the expected path. The speculative nature of this optimization is due to instructions that may be

only partially dead, but can be sunk¹ to outside the main path. This partially dead instructions define registers that are not read on the main path, but even so may not be removed since a subsequent fragment exit may use them. Bala et al. states that code sinking can be optimal as long as no compensation block must be created to hold such instructions at fragment exits. Otherwise it may lead to misprediction penalties.

2.3.2 Data Specialization

Data specialization happens whenever the compiler assumes that a certain data has a particular property, or is likely to have this property. In this paper, we distinguish three categories of data specialization: type, range and value-based. A compiler does type specialization if it assumes that the target data has a given type, and generates code customized to that type. Range specialization happens whenever the compiler assumes or infers a range of values to the data. Finally, value specialization, which we do in this paper, happens if the compiler assumes that the target data has a specific value.

2.3.3 Type Specialization

There is a large body of work related to run-time type specialization. The core idea is simple: once the compiler proves that a value belongs into a certain type, it uses this type directly, instead of resorting to costly boxing and unboxing operations. Due to the speculative nature of these optimizations, guards are inserted throughout the binary code, so that, in case the type of a variable changes, the execution environment can adapt accordingly. Rigo (2004) provides an extensive description about this form of specialization, in the context of Psyco, a JIT compiler for Python. Almási and Padua (2002) describe a complete compilation framework for Matlab that does type specialization. They assume that the type of the input values passed to a function will not change; hence, they generate binaries customized to these types. Also in the Matlab world, yet more recently, Chevalier-Boisvert et al. (2010) have used type specialization to better handle arrays and matrices.

In another front, the researchers from the Mozilla Foundation have proposed several different ways to use type information to improve JavaScript code (Chang et al. 2011, Gal et al. 2009, Hackett and Guo 2012). Gal et al. (2009)'s TraceMonkey provides a cheap inter-procedural type specialization that performs several type-based optimizations, such as converting floating point numbers to integers, and sparse objects

¹The verb “sink” is used in the compiler literature to describe the operation of moving code to side paths inside a main program path.

to dense vectors. For instance, the first optimization is done in a speculative way as soon as a variable assumes an integer value at the trace entry. If the variable type do not remains stable, it is necessary to widen its type to a double. Once identified as type instable, this variable is marked to avoid future speculative failures. Type specialization seems to be, in fact, one of the key players in the Firefox side of the browser war, as Hackett and Guo (2012) have recently demonstrated.

2.3.4 Value Range Specialization

Value Range Specialization is in the middle of the Data Specialization spectrum. Type Specialization narrows the possible values of a variable to a specific type. On the other hand, Value Specialization restricts a variable to only one value. In its turn, Value Range Specialization may assign a range of values to a variable and specialize a region of code to that specific range. Conceived by Canal et al. (2004) and based on Value Range Propagation, it is a compile-time optimization composed by three steps. The first identifies candidates for specialization. The second computes their ranges using profiling information. The third and last step, is the specialization process, which duplicates a code region and inserts tests to dynamically select the region that will be executed: either the specialized or the non-specialized one.

The typical example of range specialization are the specializers that target ranges of integer values. Perhaps the most influential work in this direction has been Bodík et al. (2000)'s ABCD algorithm. This algorithm eliminates array bounds checks on demand, and is tailored to just-in-time compilation. Bodík et al.'s algorithm relies on the "less-than" lattice Logozzo and Fähndrich (2008) to infer that some array indices are always within correct boundaries. Another work that does range specialization in JIT compilers is Sol et al. (2011)'s algorithm to eliminate integer overflow checks. This algorithm has been implemented in TraceMonkey, and has been shown to be very effective to remove these overflow tests. Sol et al.'s algorithm uses information only known at run-time to estimate intervals for the variables in the program. From these intervals it decides which arithmetic operations are always safe. Potentially unsafe operations need to be guarded, in such a way that, in face of an overflow, the 32-bit integer types can be replaced by types able to represent larger numbers.

The profitability of the specialization may be considered to reduce the number of candidates to be profiled. That is the case in the proposal of Canal et al.. Before applying the code specialization, they evaluate the benefits in expected energy savings and only then the profitable candidates are specialized. The profitability is computed through a set of mathematical formulas that take into account the cost and

the number of each generated instruction (branches, adds, comparisons, etc). Although Value Range Propagation is used in high-level code transformation, Value Range Specialization was thought to be more CPU specific, despite being compiler independent. Value Range Propagation achieved 6% of energy savings on average, and Value Range Specialization achieved 15% on average for SpecInt95 (Canal et al. 2004).

2.3.5 Value Specialization

Value Specialization, or Value-based Code Specialization, customizes the code according to specific run-time values of selected specialization variables. If the target value cannot be determined before the program runs, then it may be necessary to insert a run-time test to guard the specialized code over the possible instability of this value (Duesterwald 2005). This is the case of Profile-guided Value Specialization and Template-based Value Specialization. According to Zhang et al. (2005), Value Specialization is typically applied to procedures, which may be cloned or transformed to its typical input values that will be treated as constants. Value Specialization can also be considered as a special case of Value Range Specialization, whenever the minimum and maximal values of a range are the same. Shankar et al. (2005) lists certain classes of programs, such as interpreters, raytracers and database query executors, in which a few values are consistently used due to the execution behavior. Value Specialization, in some of these cases, can result in speedups up to 5x (Grant et al. 1999).

2.3.5.1 Template-based Value Specialization

Template-based Value Specialization combines compile-time and run-time specialization in a two-phased optimization strategy, partitioned into preprocessing and processing steps. The first is responsible to build, at compile-time, the templates that will be specialized later. The second specialize these templates according to the run-time values observed. Template-based Specialization, until the work of Consel and Noël (1996), has always been done manually (Keppel et al. 1991), where the user defines code templates or parameterized code fragments (Engler and Proebsting 1994), either with the use of a low level language or with program annotations (Auslander et al. 1996).

Keppel et al. (1993) have done an extensive study that compares traditional and a template-based compilation, demonstrating that the latter can be very competitive. The authors have studied several value specific optimizations, which rely on input variables that have constant values in certain program's regions and are used to improve the code generation. Keppel et al. analyze the trade-off between compile-time and run-

time, in the sense that the more input data to specialize the longer latency to start the execution of a program and more efficiently the specialized version will run. The benefits of such specializations can only be seized if these values remain stable enough that the optimized code is executed repeatedly. The authors have shown that these optimizations can save memory references, improve register allocation and help in dead code elimination.

The automatic generation of templates for further specialization, proposed by Consel and Noël (1996), improves the manual approach in many ways, such as portability and robustness. In the preprocessing phase, instead of manually defining code templates, the user only needs to declare a list of program invariants. Then, at compile-time, the program is analyzed to classify the variables, as either static (known) or dynamic (unknown), and to determine the program transformations to be performed. The templates are automatically generated at source-level, where the former group of variables define the overall structure of the template, whereas the variables in the latter determine the parts of this structure that must be completed dynamically. Consel and Noël presents some preliminary experiments, using variations of the *printf* function, where the specialized code runs 5 times faster than the non-specialized version. Noel et al. improves the latter work in specializing C programs and show a more complete evaluation of the proposed technique, which achieves speedups of up to 10 times.

The main difference between template-based compilation and the approach that we advocate in this work is that we do not require the pre-compilation phase in which templates are built. Using program annotations, Auslander et al. (1996) can dynamically compile regions of C programs using the knowledge that some of the variables will be invariant. They use a combination of constant propagation and constant folding to amplify the number of constants for the compiled region. Our approach is similar in the way that constant propagation and constant folding play an important role in the optimization. However, in our work, the invariant values are determined at run-time without any use of annotations.

The work of Bolz et al. (2010) is also close to ours. The authors propose to use just-in-time partial evaluation to speed up Prolog programs. Conceptually, Bolz et al.'s approach differs from ours in two main ways. First, we transform programs by using classic compiler optimizations, whereas Bolz et al. use standard partial evaluation. Second, similar to the template-based approach, Bolz et al. rely on a pre-compilation phase, in which they create hooks that the partial evaluator uses to manipulate the program. Thus, whereas we do not require a pre-compilation phase, Bolz et al. pre-process the program before running it.

2.3.5.2 Profile-guided Value Specialization

This kind of Value Specialization combines a profiler component to decide which values will be specialized in a speculative approach. This approach aims to fully automate the identification process of semi-invariant values, instead of relying on program annotations like many works on Template-based Value Specialization. Motivated by the Value Prediction technique (Gabbay and Mendelson 1996), Calder et al. (1997) have proposed a Value Profiler component able to determine if a value is invariant over the program execution. This component is also able to determine what are the most common values that each instruction can result. According to Calder et al., a Value Profiler component may be used in many optimization contexts, like code generation, adaptive execution and code specialization.

Calder et al. (1999) uses the profiler component to guide code specialization, an approach that these authors have dubbed Value-based Code Specialization. Using value profile information over SPEC 95, they were able to achieve 21% of speedup on some C programs that have high invariance on some group of values. The authors claimed that without profiling information, straight forward compiler code specialization cannot identify many of the performance bottlenecks of the considered programs. The code specialization done by Calder et al. was performed manually, since the focus of their work was in the analysis of value profiling. However, this is not a limitation inherent to this technique as showed by the work of Muth et al. (2000).

Value-profile-guided Code Specialization is another name used to refer to Profile-guided Value Specialization. Muth et al. (2000) defines it as a three-step process and also expands the profiling to expressions other than values. This process start with the identification of program points where specialization will be profitable. Once this identification is done, the value and expression profiles are built to specialize code. The work of Muth et al. describes a complete benefit analysis to guide the identification step of which value or expression to profile, in order to reduce time and space overhead. The profiler component used was based on the work of Calder et al. (1997), in spite of the fact that it does not profile memory locations. However, code specialization is done automatically instead of manually like Calder et al. (1999). This technique achieved 3.9% of speedup on SPEC 95, but with overheads of 44% for profiling and 87% for specialization, on average.

One advantage of pure Speculative Value Specialization, without a profiler, is that it does not incur in high overheads like these. Such technique relies on heuristics that determine when the code should be specialized. However, imprecisions in these heuristics may increase the number of recompilations, as we show in Chapter 4. On the

Profile-guided Value Specialization side, the use of a dedicated hardware can alleviate this problem without implying software overhead. Zhang et al. (2005) have empirically evaluated this approach, obtaining a 20% speedup in a small suite of benchmarks. However, the profiler component proposed is only available in a simulated environment and is not present on real processors.

2.3.5.3 Speculative Value Specialization

Speculative Value Specialization, until now and as far as we know, was only applied with the use of a profiler component. In hardware, it can be used to improve conventional trace cache architectures (Zhang et al. 2007). Zhang et al. have obtained a speedup of 17% over conventional hardware value prediction, mostly due to the reduction in the length of dependence chains. This speedup has allowed them to improve value locality. The technique that they presented uses semi-invariant load values detected at run-time to dynamically specialize the program's code. This approach relies on a profiler component implemented in hardware to detect semi-invariant loads. This strategy is inherently speculative, as these semi-invariant loads may change over time, in which case the system will need to recover from mispredictions.

Our first attempt to implement Speculative Value Specialization without a profiler component is described in Alves et al. (2012). That endeavor was a very limited study: the engine execution was aborted if a specialized function was called more than once. In that work we also only implemented Constant Propagation, besides Parameter Specialization. Thus, we could only test our ideas in an artificial set of benchmarks. Nevertheless, this preliminary study has provided the groundwork to a more robust and functional implementation (Costa et al. 2013), as well as a more complete evaluation. The present work summarizes all these contributions and provides complementary analysis to the presented results.

2.3.6 Input-centric compilation

Input-centric compilation is a general code generation method similar to Value Range Specialization. The first work in this line has been proposed by Canal et al. (2004). Canal et al. generate multiple code regions for the same program, each of them specialized for a particular range of input values. Dynamic tests decide, at run-time, which version of the program is the most appropriate to deal with a given input. A similar work has been developed by Tian et al. (2010, 2011). In this model, the compiler generates the best possible code for a certain universe of possible inputs. Machine learning might be used to determine which strategy the compiler should adopt to produce code.

A recent example of input-oriented compilation is the work of Samadi et al. (2012). The authors generate programs containing distinct sub-programs to handle different kinds of inputs. Run-time characteristics of the particular input determine which routine will be activated. The input aware compiler has the advantage of being more general than our approach, because it generates code that works on different inputs. Our code only works with certain values. However, the optimizations that we can perform are more extensive: we trade generality for over-specialization.

2.4 Discussion

A great body of research about Speculative Value Specialization relies on profiling to identify profitable code to specialize. A profiler component can reduce the number of mispredictions through an invariance analysis of values to be specialized. However, the overhead of such analysis can limit their use in some environments, like embedded systems and Just-in-Time compilation for Web or mobile devices. Such environments can have strict limits to memory used or user response time. This overhead can be reduced through the use of a specialized hardware, as proposed by (Zhang et al. 2007). However, this hardware is only available through simulated environments. In that context, Speculative Value Specialization without the use of a profiler may present a better trade-off between compiler analysis time and user experience latency. This is the opportunity seized in this dissertation, where this concept is applied to improve the performance of JavaScript on the Web.

Chapter 3

Parameter-based Value Specialization

This chapter explains what is Parameter-based Value Specialization and how it works. The first part of this chapter presents the empiric study that motivated this work. We start describing the methodology used in the experiments. Then we present an analysis of the function call behavior of real world JavaScript programs. The second part of the chapter describes parameter specialization and how it works on an industrial JIT compiler. The third, and last part, revisits classic optimizations highlighting how they can benefit from parameter specialization.

3.1 Motivation

3.1.1 Methodology

This work was motivated by the empiric observation that most JavaScript functions are called only once, or are always called with the same arguments. We rely on the methodology used by Richards et al. (2010) to build the experiments presented in this chapter. As Richards et al. has pointed out, the most popular benchmarks may not represent correctly the behavior of real JavaScript programs. Therefore, we analyze the behavior of popular websites and compare them with the behavior of the JavaScript benchmarks available. The list of websites used in all experiments was retrieved from the Alexa index (Alexa Internet Inc 2010). The benchmarks analyzed were: SunSpider, Kraken and V8. All experiments were done using a modified version of Mozilla Firefox browser, instrumented to collect the necessary information for each experiment.

Richards et al. (2010) instrumented the Webkit (Apple Inc. 2012b) web browser to collect JavaScript execution traces of 103 different real-world web sites. To mimic a typical user interaction several traces were recorded during the navigation of each of these sites, and they were averaged in the metrics. The analysis developed by Richards et al., performed offline, combines both the recorded traces and source code static metrics. The trace recording contains a mixture of interpreter operations, like reads and calls, and browser events, like garbage collection and eval invocations where the evaluated string is also recorded. Some static analysis were done using the Rhino JavaScript compiler (Mozilla Foundation 2012c), such as the code size metrics. One important result obtained by Richards et al. is about the call site dynamism which the analysis reveals as being highly monomorphic: 81% of call sites always invokes the same method. This is an important result, because it justifies the speculative inlining of methods.

In this work, we focus on the JavaScript behavior related with function calls. For each script, we collect information about all function calls invoked during the execution. We log the following information about each function called: length in bytecodes, name (including script name and line number of the function definition) and the value plus type of actual arguments. To achieve meaningful results from the real websites, our script imitates a typical user session as much as possible. We simulate this mock user session using the jQuery (jQuery Foundation 2012) API to build a script, which collects all links and buttons of a webpage and randomly executes them to simulate mouse events. Keyboard interaction was achieved by collecting all input fields of the current webpage, and filling them with random strings. We have manually navigated through some of these webpages, to certify that our robot produces results that are similar to those that would be obtained by a human being.

3.1.2 Function call behavior

The histogram in Figure 3.1 shows how many times each different JavaScript function is called. This histogram clearly delineates a power distribution. In total we have seen 23,002 different JavaScript functions in the 100 visited websites. 48.88% of all these functions are called only once during the entire browser session. 11.12% of the functions are called twice. The most invoked functions are from the Kissy UI library, located at Taobao content delivery network (<http://a.tbcdn.cn>), and Facebook JavaScript library (<http://static.ak.fbcdn.net>), respectively. The first one is called 1,956 times and the second 1,813 times. These numbers show that specializing functions to the run-time value of their parameters is a reasonable approach in the JavaScript

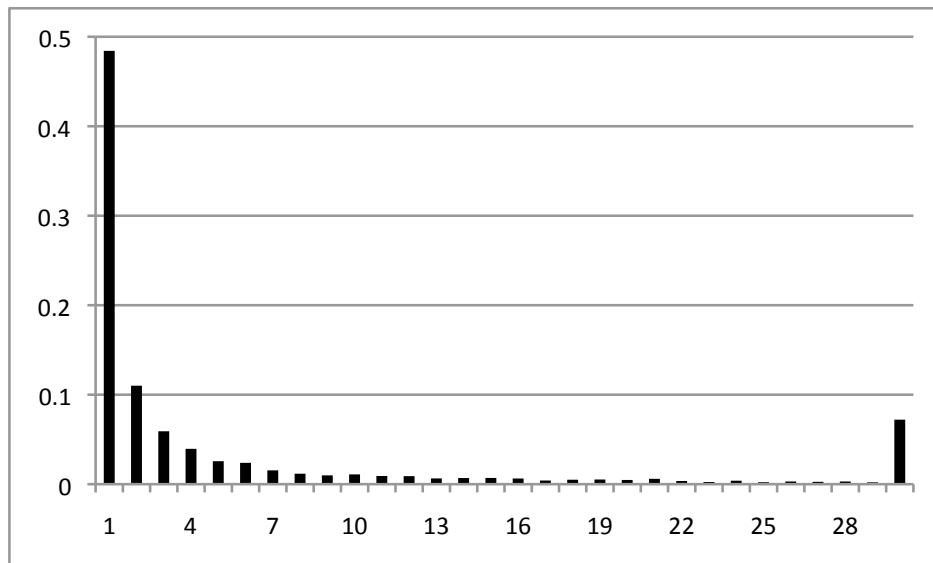


Figure 3.1: Histogram showing the percentage of JavaScript functions (Y-axis) that are called n times (X-axis). Data taken from Alexa’s 100 top websites. The histogram has 353 entries; however, we only show the first 29. The tail has been combined in entry 30.

world.

If we consider functions that are always called with the same parameters, then the distribution is even more concentrated towards 1. The histogram in Figure 3.2 shows how often a function is called with different parameters. This experiment shows that 59.91% of all the functions are always called with the same parameters. The descent in this case is impressive, as 8.71% of the functions are called with two different sets of parameters, and 4.60% are called with three. This distribution is more uniform towards the tail than the previous one: the most varied function is called with 1,101 different parameters, the second most varied is called with 827, the third most with 736, etc. If it is possible to reuse the same specialized function when its parameters are the same, the histogram in Figure 3.2 shows that the speculation that we advocate in this work has a 60% hitting rate. We keep a cache of actual parameter values, so that we can benefit from this regularity. Thus, if the same function is called many times with the same parameters, then we can still run its specialized version.

We have also built these histograms to popular JavaScript benchmarks. These results are given in Figure 3.3. The new histograms are more varied than in the previous analysis. We speculate that this greater diversity happens because we are considering a universe with much less elements: We have 154 distinct functions in SunSpider, 186 in Kraken, and 320 in Google’s V8. Nevertheless, we can still observe

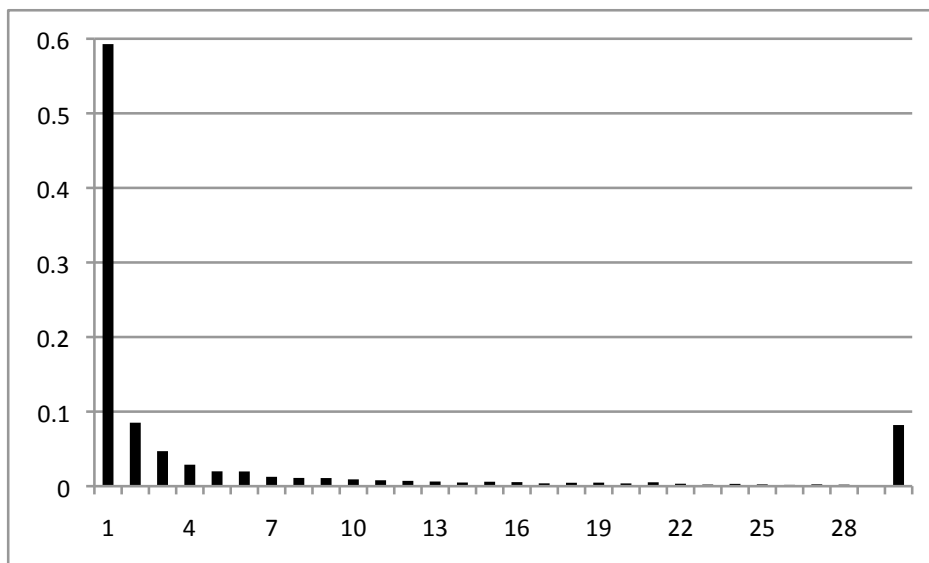
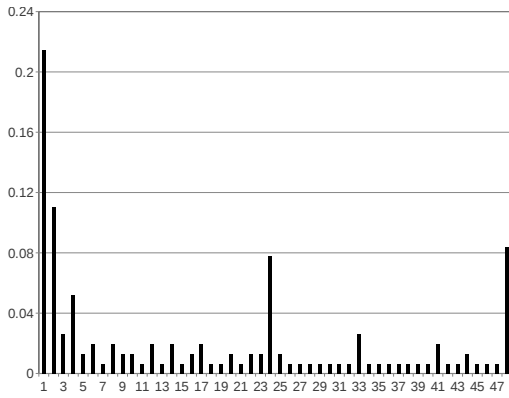


Figure 3.2: Histogram showing the percentage of JavaScript functions (Y-axis) that are called with n different sets of arguments (X-axis). Data taken from Alexa’s 100 top websites. We only show the first 30 entries.

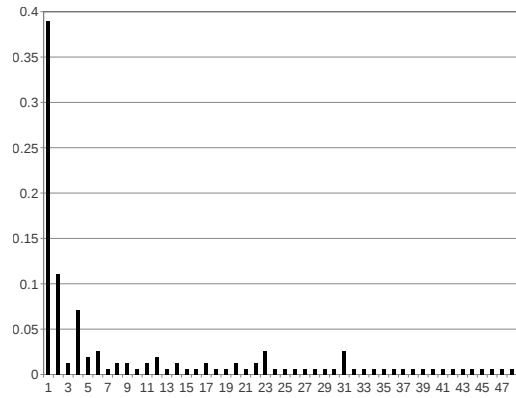
a power law, mainly in SunSpider’s and Kraken’s distribution. 21.43% of SunSpider’s functions are called only once. This number is only 4.68% in V8, but is 39.79% in Kraken. The function most often called in SunSpider, `md5_ii` from the `crypto-md5` benchmark, was invoked 2,300 times. In V8 we have observed 3,209 calls of the method `sc_Pair` in the `earley-boyer` benchmark. In Kraken, the most called function is in `stanford-crypto-ccm`, an anonymous function invoked 648 times.

If we consider how often each function is invoked with the same parameters, then we have a more evident power distribution. Figure 3.3 (Bottom) shows these histograms. We have that 38.96% of the functions are called with the same actual parameters in SunSpider, 40.62% in V8, and 55.91% in Kraken. At least for V8 we have a stark contrast with the number of invocations of the same function: only 4.68% of the functions are called a single time, yet the number of functions invoked with the same arguments is one order of magnitude larger. In the three collections of benchmarks, the most called functions are also the most varied ones. In SunSpider, each of the 2,300 calls of the `md5_ii` function receives different values. In V8 and Kraken, the most invoked functions were called with 2,641 and 643 different parameter sets, respectively.

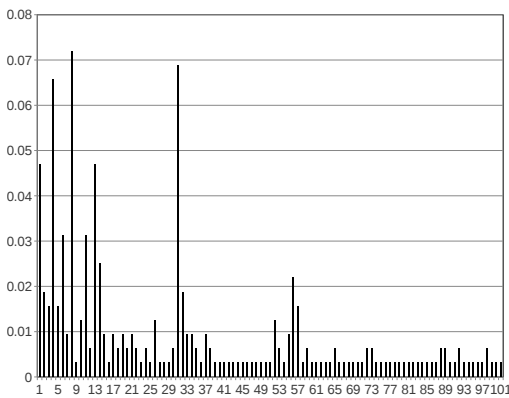
The histograms in Figure 3.4 shows a similar analysis of function calls on the three benchmark suites, but now considering only functions that IonMonkey compiles during the execution of each suite. Only functions that executes for a long time are represented in the charts. The charts (b), (d) and (f) only represent functions that have



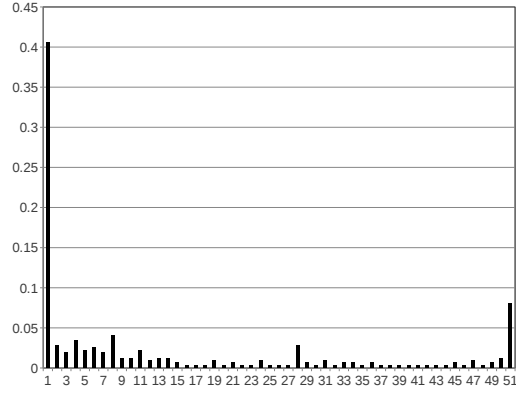
(a) Function calls on SunSpider



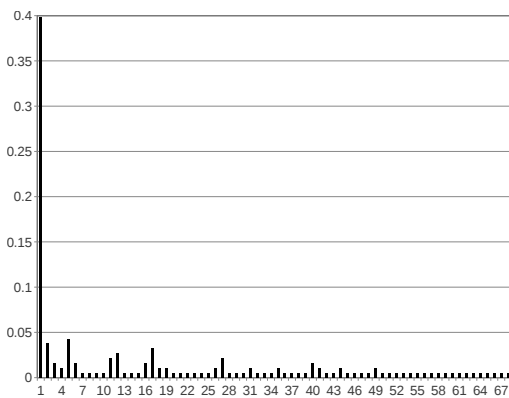
(b) Function calls with same parameters on SunSpider



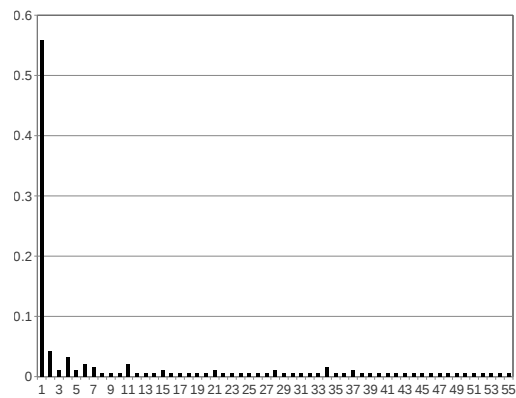
(c) Function calls on V8



(d) Function calls with same parameters on V8

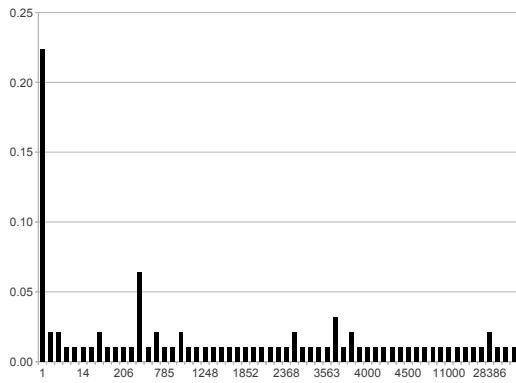


(e) Function calls on Kraken

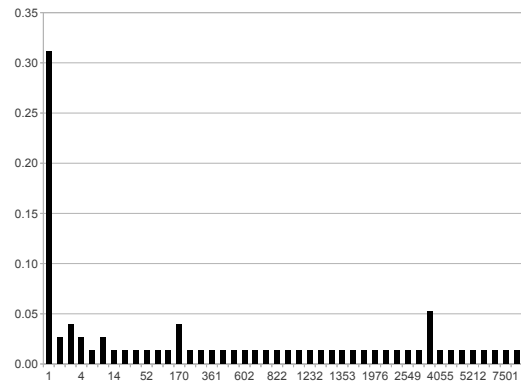


(f) Function calls with same parameters on Kraken

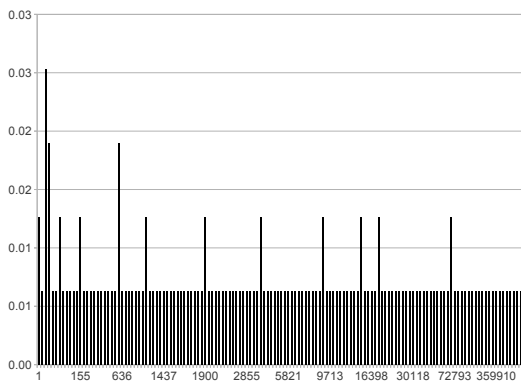
Figure 3.3: Invocation histograms for three different benchmark suites. (Left) Fraction of the number of times that each function is called. (Right) Fraction of the number of times that each function is called with the same parameters.



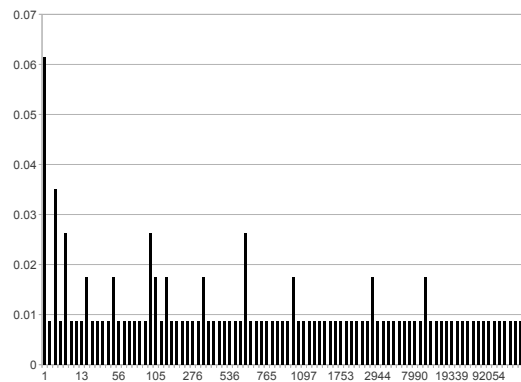
(a) Compiled function calls on SunSpider



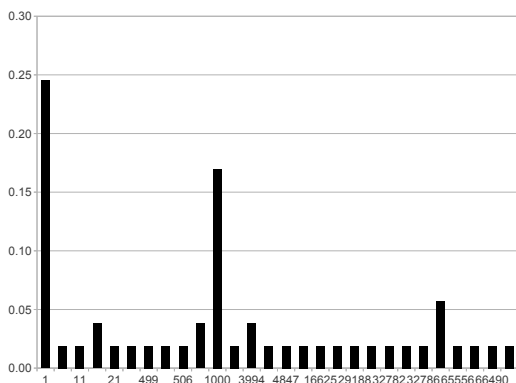
(b) Compiled function calls with same parameters on SunSpider



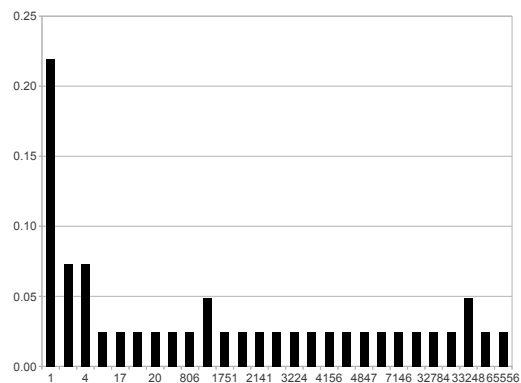
(c) Compiled function calls on V8



(d) Compiled function calls with same parameters on V8



(e) Compiled function calls on Kraken



(f) Compiled function calls with same parameters on Kraken

Figure 3.4: Invocation histograms for three different benchmark suites considering only functions compiled by IonMonkey. (Left) Fraction of the number of times that each function is called. (Right) Fraction of the number of times that each function is called with the same parameters.

Benchmark	Calls	ZeroArgs	Functions	Compiled	CompiledZeroArgs
SunSpider-1.0	494907	8.62%	2269	4.14%	0.79%
V8 version 6	9472644	26.86%	1744	9.06%	2.86%
Kraken 1.1	615965	6.49%	116	45.69%	11.20%
Alexa	1348443	8.27%	61435	0.01%	0.00%

Table 3.1: Function Analysis Statistics

(Calls) Number of function calls that have been executed for each benchmark,
 (ZeroArgs) Percentage of function calls that do not have arguments,
 (Functions) Number of functions analyzed in each benchmark,
 (Compiled) Percentage of functions compiled by IonMonkey,
 (CompiledZeroArgs) Percentage of functions compiled by IonMonkey that do not have arguments.

at least one parameter. Table 3.1 presents statistics about the function calls analyzed. In SunSpider, only 8.62% of the function calls do not have arguments. In Kraken, this percentage is even smaller, only 6.49% of function calls do not have arguments. However, in V8 this percentage is more representative (26.86%), which reflects the histograms (c) and (d) of Figure 3.4. SunSpider and V8, have a similar number of compiled functions and compiled functions that do not have arguments. In Kraken, the number of compiled function is 45%. However, the percentage of compiled functions that do not have arguments represent 11.20% of all functions of this benchmark. For the Alexa index, almost none (0.01%) of the called functions are compiled by IonMonkey. The JavaScript functions found in Alexa are quite simple and do not represent intensive JavaScript webpages found in the wild.

3.1.3 The types of the parameters

We have performed a comparison between the types of the parameters used by functions called with only one set of arguments in the benchmarks, and in the Alexa top 100 websites. The results of this study are shown in Figure 3.5. Firstly, we observe great diversity between the benchmarks and the web, and among the benchmarks themselves. Nevertheless, one fact is evident: the benchmarks use integers much more often than the JavaScript functions that we found in the wild. 37.5%, 48.72% and 33.03% of the parameters used by functions in SunSpider, V8 and Kraken are integers. On the Internet, only 6.36% of the parameters are integers. In this case, objects and strings are used much more often: 35.57% and 32.95% of the time. Some of the optimizations that we describe in this work, notably constant propagation, can use integers, doubles and

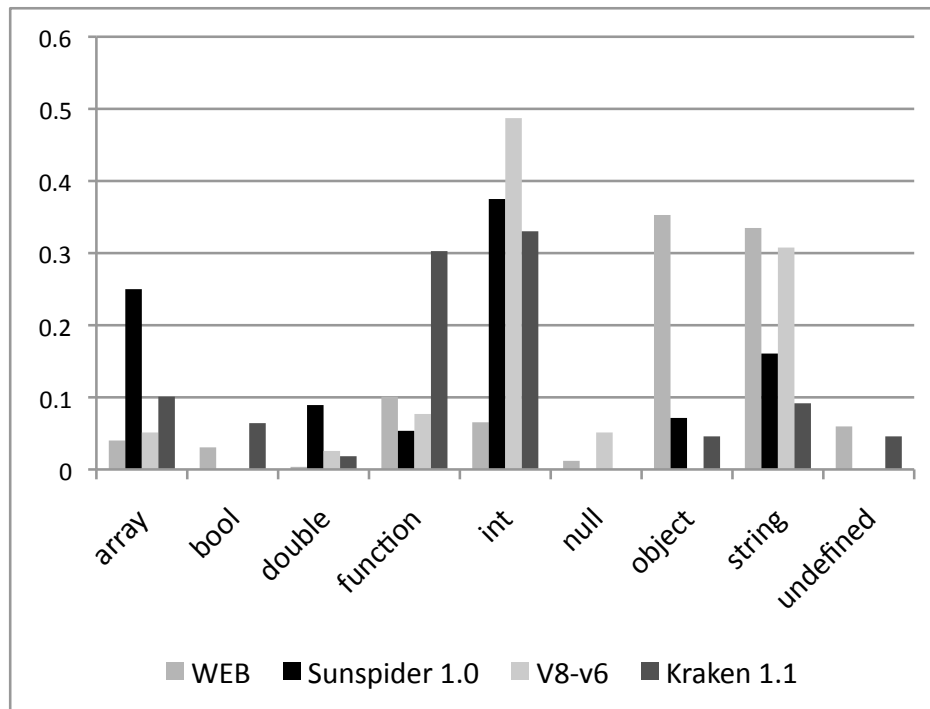


Figure 3.5: The most common types of parameters used in benchmarks and in actual webpages.

booleans with great benefit: these primitive types allow us to solve some arithmetic operations at code-generation time. We can do less with objects, arrays and strings: we can inline some properties from these types, such as the `length` constant used in strings. We can also solve some conditional tests, e.g., `==`, `===`, etc, and we can solve calls to the `typeof` operator.

3.2 Parameter Based Speculative Value Specialization

Our idea is to replace the parameters of a function with the values that they hold when the function is called. Before explaining how we perform this replacement, we will briefly review the life cycle of a program executed through a just-in-time compiler. In this work, we focus on the binaries generated by a particular compiler – IonMonkey – yet, the same code layout is used in other JIT engines that combine interpretation with code generation.

Some JavaScript run-time environments, such as Chromium’s V8, compile a function the first time that it is called. Other run-time systems compile code while this code

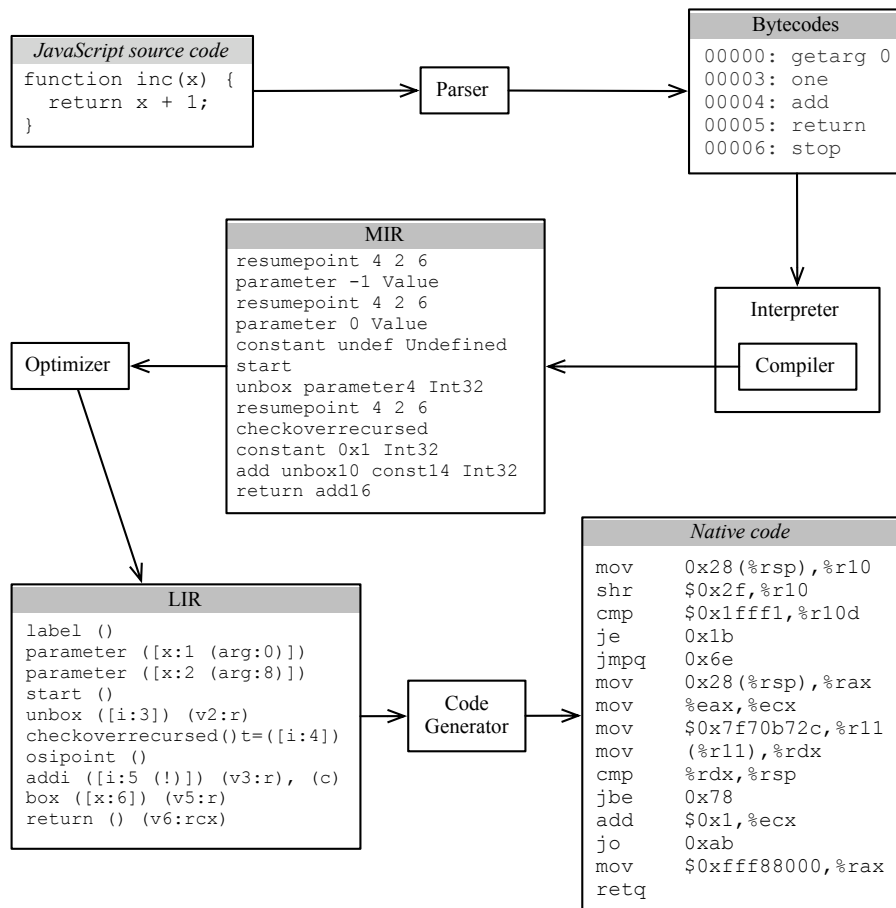


Figure 3.6: The life cycle of a JavaScript program in the SpiderMonkey/IonMonkey execution environment.

is interpreted. The Mozilla Firefox engine follows the second approach. A JavaScript function is first interpreted, and then, if a heuristics deems this function worth compiling, it is translated to native code. Figure 3.6 illustrates this interplay between interpreter and just-in-time compiler. Mozilla’s *SpiderMonkey* engine comes with a JavaScript interpreter. There exists several JIT compilers that work with this interpreter, e.g., TraceMonkey (Gal et al. 2009), JägerMonkey (Hackett and Guo 2012) and IonMonkey. We will be working with the latter.

The journey of a JavaScript function in the Mozilla’s Virtual Machine starts in the parser, where the JavaScript code is transformed into bytecodes. These bytecodes form a stack-based instruction set, which SpiderMonkey interprets. Some JavaScript functions are either called very often, or contain loops that execute for a long time. We say that these functions are *hot*. Whenever the execution environment judges a

function to be hot, this function is sent to IonMonkey to be translated to native code.

The JavaScript function, while traversing IonMonkey’s compilation pipeline, is translated into two intermediate representations. The first, the *Middle-level Intermediate Representation*, or MIR for short, is the baseline format that the compiler optimizes. MIR instructions are three-address code in the static single assignment (SSA) form (Cytron et al. 1991). In this representation we have an infinite supply of *virtual registers*, also called variables. The main purpose of the MIR format is to provide the compiler with a simple representation that it can optimize. One of the optimizations that IonMonkey performs at this level is global value numbering; a task performed via the algorithm first described by Alpern et al. (1988). It is at this level that we apply the optimizations that we describe in this section.

The optimized MIR program is converted into another format, before being translated into native code. This format is called the *Low-level Intermediate Representation*, or LIR. Contrary to MIR, LIR contains machine-specific information. MIR’s virtual registers have been replaced by a finite set of names whose purpose is to help the register allocator find locations to store variables. After IonMonkey is done with register allocation, its code generator produces native binaries. SpiderMonkey diverts its flow to the address of this stream of binary instructions, and the JavaScript engine moves to native execution mode.

This native code will be executed until either it legitimately terminates, or some guard evaluates to false, causing a *recompilation*. Just-in-time compilers usually speculate on properties of run-time values in order to produce better code. IonMonkey, for instance, uses type specialization. JavaScript represents numbers as infinitely large floating-point values. However, many of these numbers can be represented as simple integers. If the IonMonkey compiler infers that a numeric variable is an integer, then this type is used to compile that variable, instead of the more expensive floating-point type. On the other hand, the type of this variable, initially an integer, might change during the execution of the JavaScript program. This modification triggers an event that aborts the execution of the native code, and forces IonMonkey to recompile the entire function, boxing the variable as a floating-point value.

3.2.1 The Anatomy of a MIR program

Figure 3.7 shows an example of a control flow graph (CFG) that IonMonkey produces. In the rest of this work we will be using a simplified notation that represents the MIR instruction set. Contrary to a traditional CFG, the program in Figure 3.7 has two entry points. The first, which we have labeled *function entry point* is the path taken

whenever the program flow enters the binary function from its beginning. This is the path taken whenever a function already compiled is invoked. The second entry point, the *on stack replacement* (OSR) block, is the path taken by the program flow if the function is translated into binary during its interpretation. As we have mentioned before, a function might be compiled once some heuristics in the interpreter judges that it will run for a long time. In this case, the interpreter must divert the program flow directly to the point that was being interpreted when the native code became active. The OSR block marks this point, usually the first instruction of a basic block that is part of a loop.

The CFG in Figure 3.7 contains several special instructions called `resumepoint`. These instructions indicate places where the state of the program must be saved, so that if it returns to interpretation mode, then the interpreter will not be in an inconsistent state. Resume points are necessary after function calls, for instance, because they might have side effects. On the other hand, referentially transparent commands do not require saving the program state back to the interpreter.

3.2.2 Parameter Specialization

The core optimization that we propose in this work is *parameter specialization*. This optimization consists in replacing the arguments passed to a function with the values associated with these arguments at the time the function is called. Our optimizer performs this replacement while the MIR control flow graph is built; therefore, it imposes **zero overhead** on the compiler. That is, instead of creating a virtual name for each parameter in the graph, we create a constant with that parameter's run-time value. We have immediate access to the value of each parameter, as it is stored in the interpreter's stack. There are two types of inputs that we specialize: those in the function entry block, and those in the OSR block. Figure 3.8 shows the effects of this optimization in the program first seen in Figure 3.7.

3.3 Revisiting Classic Optimizations

3.3.1 Constant Propagation

The substitution of arguments by constants improves, by itself, the performance of the binaries that we specialize. However, its use combined with constant propagation can provide opportunities for other optimizations, such as bound check elimination and dead code elimination. Constant propagation is, possibly, the most well-known

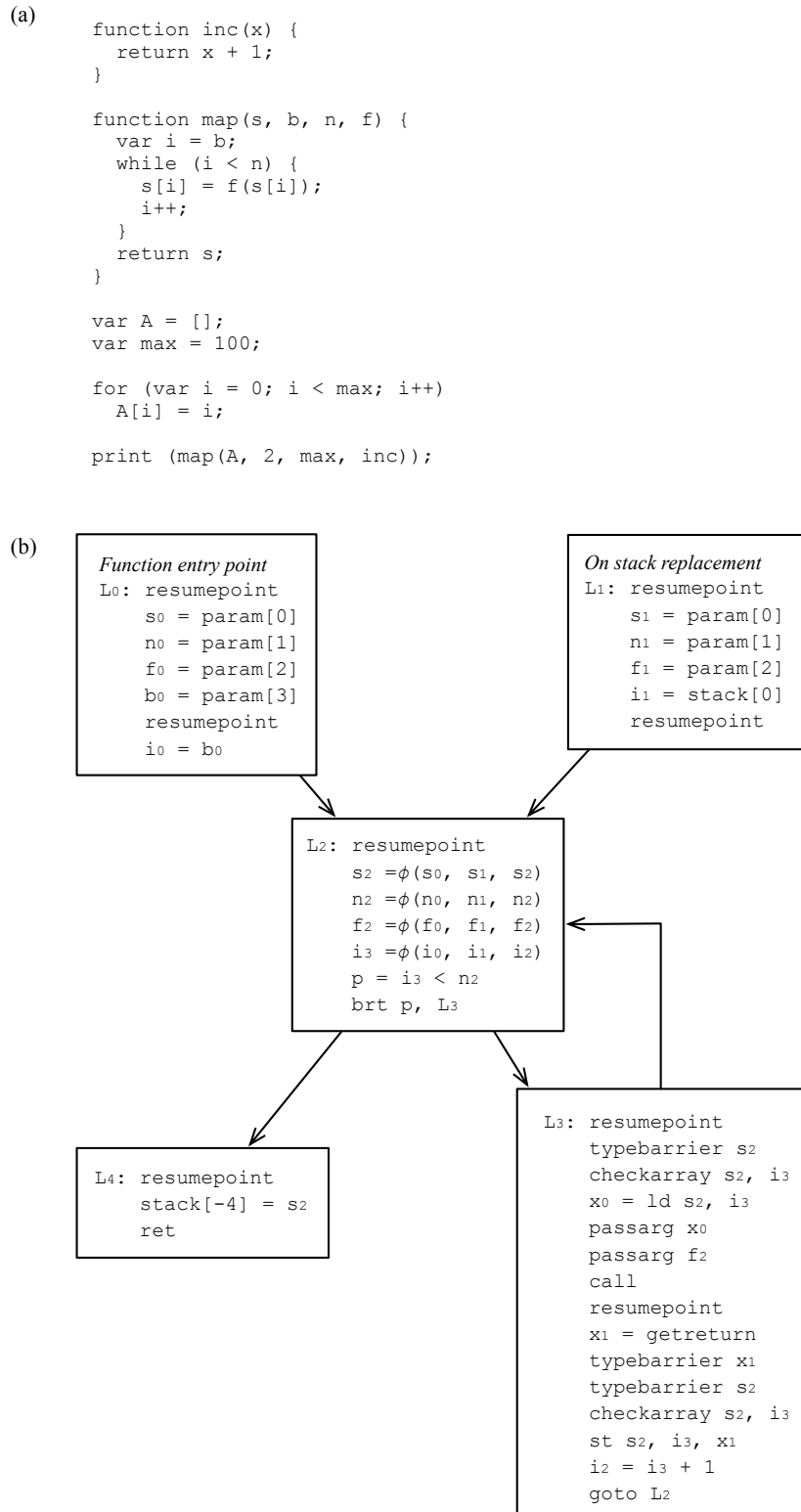


Figure 3.7: (a) The JavaScript program that we will use as a running example. (b) The control flow graph of the function `map`.

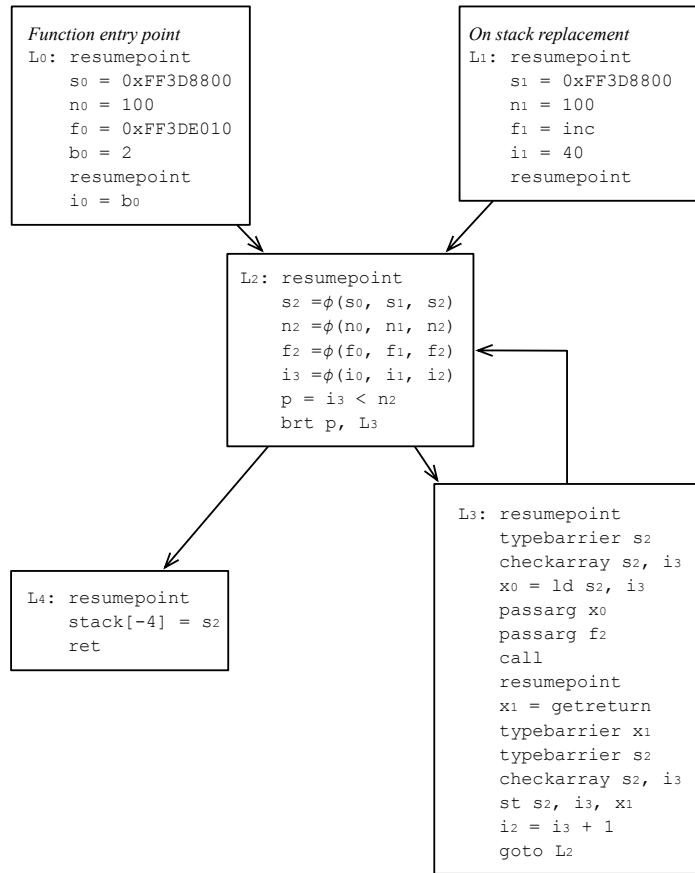


Figure 3.8: The result of our parameter specialization applied onto the program in Figure 3.7.

code optimization, and it is described in virtually every compiler textbook. We have implemented the algorithm present in Aho *et al.*'s classic book (Aho et al. 2006, p.633-635). Each program variable is associated with one element in the lattice $\perp < c < \top$, where c is any constant. We iterate successive applications of a meet operator until we reach a fixed point. This meet operator is defined as $\perp \wedge c = c$, $\perp \wedge \top = \top$, $\top \wedge c = \top$, $c_0 \wedge c_1 = c_0$ if $c_0 = c_1$ and $c_0 \wedge c_1 = \top$ otherwise. We have opted for the simplest possible implementation of constant propagation, to reduce the time overhead that our optimization imposes on the run-time environment. Thus, contrary to Wegman and Zadeck (1991) seminal algorithm, we do not extract information from conditional branches.

Figure 3.9 shows the code that results from the application of constant propagation on the program seen in Figure 3.8. If all the arguments of an instruction i are constants, then we can evaluate i at compilation time. If i defines a new variable v , then we can replace every use of v by the constant that we have just discovered. The

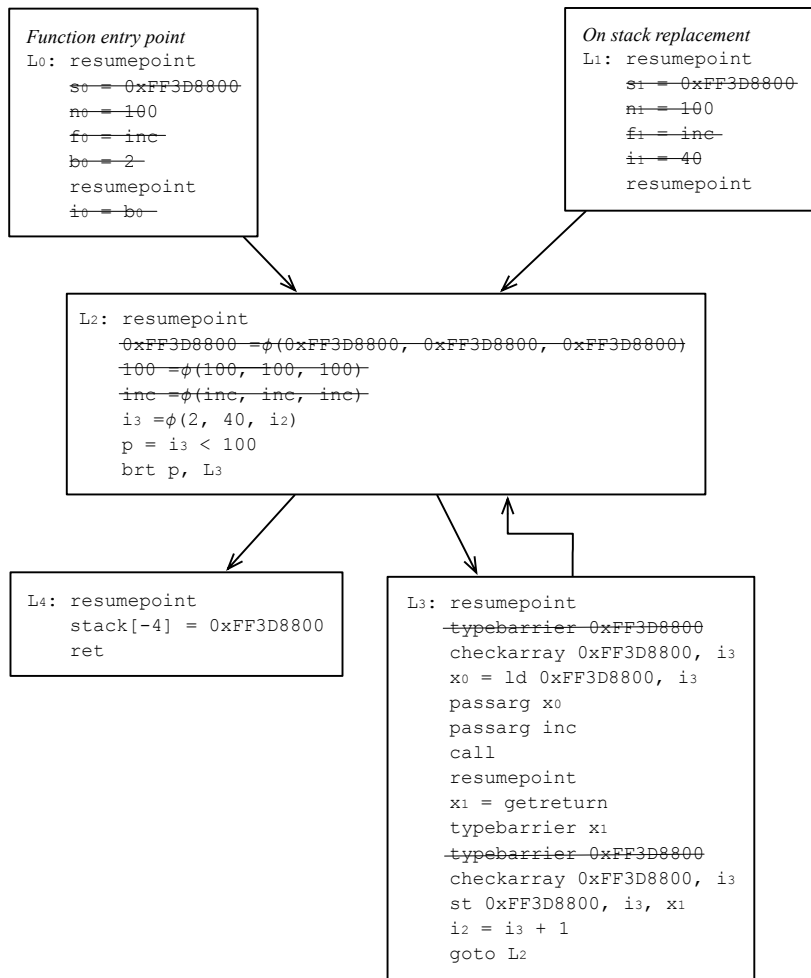


Figure 3.9: The result of applying constant propagation on the program in Figure 3.8.

elimination of an instruction that only operates on constants is called *folding*. We have marked the 14 instructions that we have been able to fold in Figure 3.9. We can fold many JavaScript typical operations. Some of these operations apply only to primitive types, such as numbers, e.g., addition, subtraction, etc. Others, such as the many comparison operators, e.g., `==`, `!=`, `===`, `!==` and the `typeof` operator, apply on aggregates too.

JavaScript is a very reflective language, and run-time type inspection is a common operation, not only at the development level, but also at the code generation level. As an example, in Figure 3.7 we check if `s` is an array, before accessing some of its properties. Our constant propagation allows us to fold away many type guards, which are ubiquitous in the code that IonMonkey generates. We have folded the two type guards in block `L3`. This optimization is safe, as there is no assignment to variable `s` in the entire function.

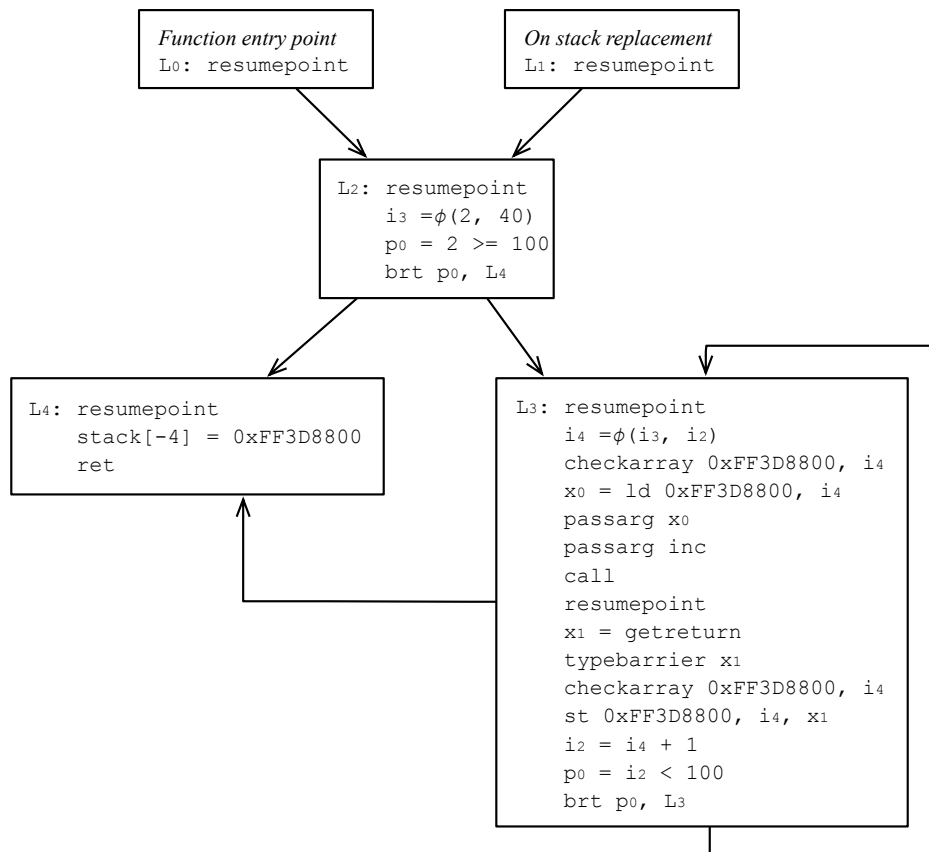


Figure 3.10: The result of applying loop inversion on the program seen in Figure 3.9.

3.3.2 Loop Inversion

Loop inversion is a classic compiler optimization that consists in replacing a while loop by a repeat loop. The main benefit of this transformation is the replacement of a conditional and an unconditional jump inside a loop with only a conditional loop at its end. Figure 3.10 shows the result of performing loop inversion in the program seen in Figure 3.9. Usually loop inversion inserts a wrapping conditional around the repeat loop, to preserve the semantics of the original program. This conditional, only traversed once by the program flow, ensures that the body of the repeat loop will not be executed if the corresponding while loop iterates zero times. Loop inversion does not directly benefit from the knowledge of run-time values. However, we have observed that a subsequent dead-code elimination phase can remove the wrapping conditional. This elimination is possible because our parameter specialization often lets us know that a loop will be executed at least once.

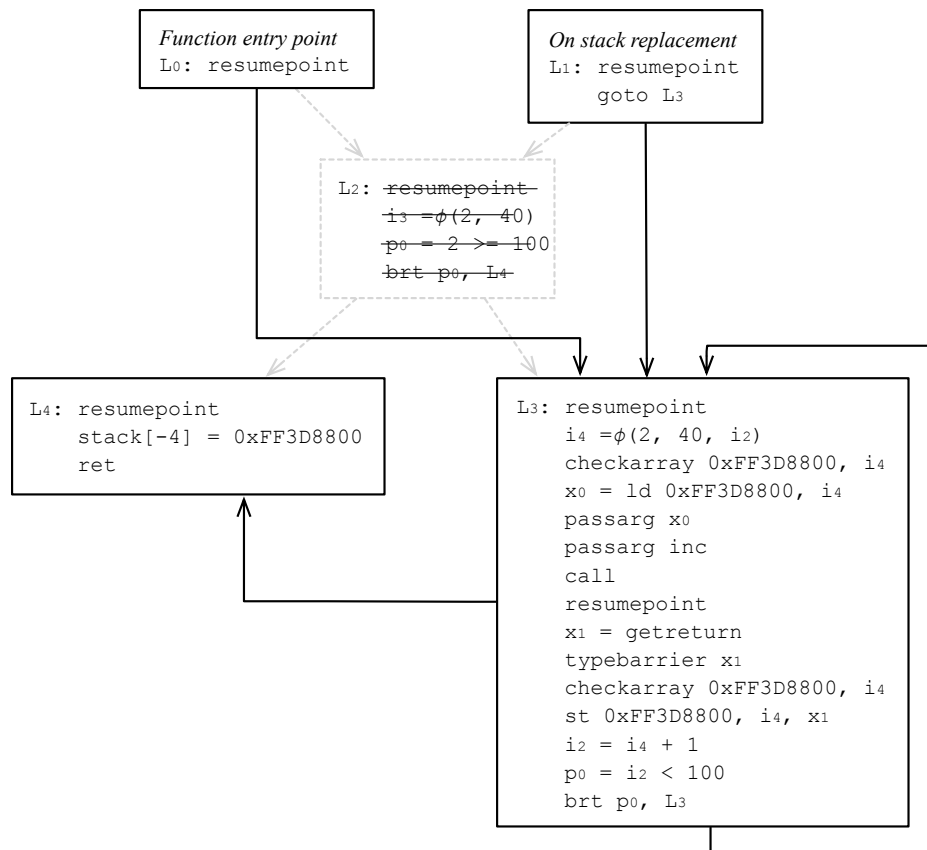


Figure 3.11: The result of applying dead-code elimination on the program seen in Figure 3.10.

3.3.3 Dead-Code Elimination

Dead-code elimination removes instructions that we prove that cannot be reached by the program flow. We run it after constant propagation, in order to give instruction folding the chance to transform conditional branches into simple boolean values. Whenever this extensive folding is possible, the outcome of the conditional branch can be predicted at compile-time; thus, we can safely remove the branch instruction and, possibly, blocks of unreachable code. Figure 3.11 shows the effects of dead-code elimination on the program in Figure 3.10. We have removed block L_2 , because the result of the comparison inside this block is known at code generation time. Notice that we keep the function entry block. We only keep this block because we can cache the binaries that we produce, in case a function is called with the same parameters again. If a function compiled to native code is called again, then execution must start at the function entry point.

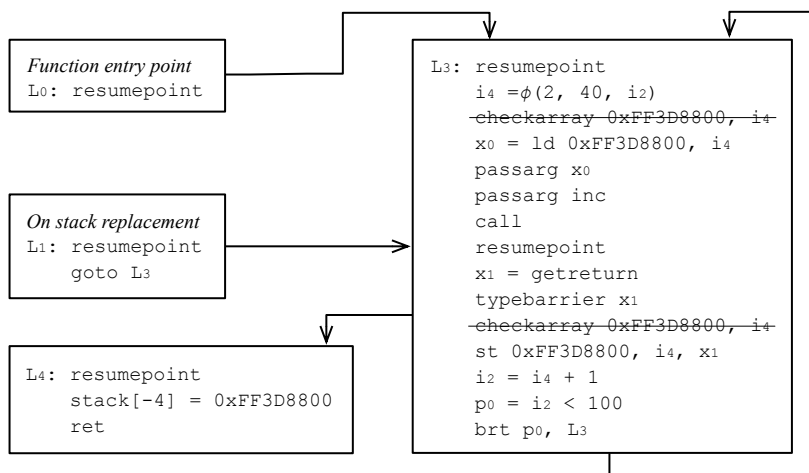


Figure 3.12: The result of eliminating array bounds checks from the program shown in Figure 3.11.

3.3.4 Array Bounds Check Elimination

JavaScript is a type safe language, which means that any value can only be used according to the contract specified by its run-time type. As a consequence of this type safety, array accesses in JavaScript are bounds checked. Accesses outside the bounds of the array return the `undefined` constant, which is the only element in the Undefined data type. Bounds checking an index i is a relatively expensive operation, because, at the native code level, it requires loading the array length property l , and demands two conditional tests: $i \geq 0$ and $i < l$. The knowledge of function inputs allows us to eliminate some simple bounds checks.

To perform this optimization, we need to identify integer variables that control loops. If these induction variables are bounded by a known value, then we can perform a trivial kind of range analysis to estimate the minimum and maximum values that array indices might receive. In order to keep our optimizer simple and efficient, we only recognize function variables defined by the pattern $i_0 = exp; i_1 = \phi(i_0, i_2); i_2 = i_1 + c_2$. Variables i_3 and i_4 , plus the constant 2, in Figure 3.11(a) follow this pattern. Variable i_2 is initially assigned the constant 2, and $i_2 < 100$ inside the loop; hence, its range is $[2, 99]$. Moreover, $i_4 = \phi(2, i_2)$; thus, its range is also $[2, 99]$. Therefore, any access of array s_2 , e.g., reference `0xFF3D8800` in the figure, indexed by i_4 is safe, as s_2 's length is 100. Figure 3.12 shows the result of eliminating the bounds checks from the program in Figure 3.11.

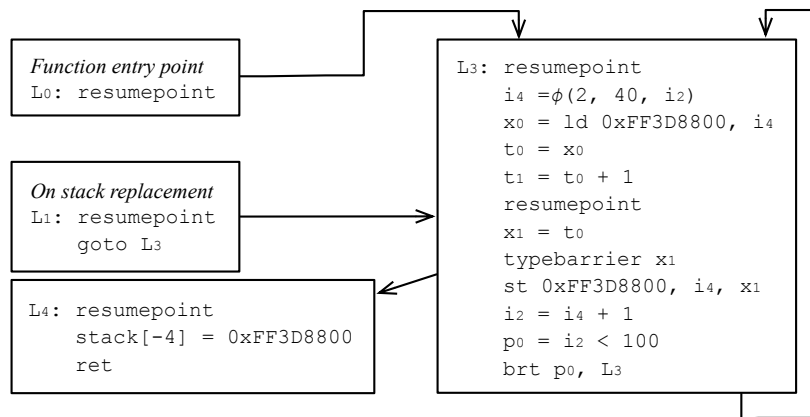


Figure 3.13: The result of inlining the `inc` function in the code presented by Figure 3.12.

3.3.5 Function Inlining

JavaScript supports closures; therefore, it is possible to pass a function as an argument to another one. We inline functions passed as arguments, whenever possible. Figure 3.13 shows the result of replacing the call to function `inc`, seen in Figure 3.12, by its body. IonMonkey already performs function inlining; however, it does it much later than we do. IonMonkey’s inliner is profile-guided. Once a function is called 10,240 times, it decides to inline it. Closures are not immediately inlined, as they are passed as formal parameters to a function that can be called with many different actual parameters. Furthermore, inlining closures requires guards: if the host function is called again, this time with a different closure, recompilation must take place. Our aggressive approach to inlining avoids all this burden. We inline a closure as soon as we compile the host function, and we do not use guards. In case the function is called again, our entire code will be discarded; hence, these guards would not be necessary.

3.3.6 Other Optimizations

There are many classic compiler optimizations that we have not considered in this work, either due to the lack of time, or due to technological limitations in the current implementation of IonMonkey. Two of these optimizations that we plan to investigate in the future are loop unrolling and integer overflow check elimination. We speculate that loop unrolling can be very effective in our scenario, as we can use the simple analysis of Section 3.3.4 to find out how many times most of the loops will iterate.

3.4 Discussion

This chapter presented our optimization and how it can be used to enhance many classic optimizations, which may have inexpressive results when applied on dynamic languages. In this chapter we have also observed that many JavaScript functions are called most of the time with the same parameters. This is the key observation that motivates our work. Although popular JavaScript benchmarks often do not represent well the behavior of real world scripts (Richards et al. 2010), the Figure 3.3 shows that they could be used to evaluate the proposed optimization, as presented in the next chapter.

Chapter 4

Experiments

In this chapter we present an evaluation of the Parameter-based Speculative Value Specialization over popular benchmarks and a few real world web pages. We start by presenting run time results for different combinations of the optimizations described in Section 3.3 for SunSpider, V8 and Kraken. Then we analyse the overhead of our optimization regarding the size of generated code, the compilation overhead and the number of recompilations. All experiments that we describe in this chapter were performed on a Quad-Core Intel i5 processor with 3.3GHz of clock and 8GB of RAM running Ubuntu 12.04.1 32-bits. The IonMonkey version that we are using as baseline was obtained from the Mozilla repository¹ on August 3rd, 2012.

4.1 Benchmarks

Timing JavaScript applications in actual webpages is not trivial, because too many factors, mostly related to I/O operations, have an impact on the run time of these programs. Therefore, we chose to use three well know benchmarks: SunSpider, Kraken and V8 in our experiments. The advantage of the benchmarks is that we can measure their execution time reliably. The disadvantage is that, as pointed by Richards et al. (2010), benchmarks might not reflect the true nature of the JavaScript applications found in the wild. The data that we shown in Section 3.1 is a testimony of this fact. On one hand, Figure 3.5 shows that real world applications tend to use instances of `object` more often than instances of simpler types. The optimizations in Section 3.3 work better in the latter case. On the other hand, Figures 3.1, 3.2 and 3.3 seem to imply that functions tend to be called with the same parameters more often in the wild, then in benchmarks. To mitigate these inconsistencies, we will also use the benchmarks

¹<http://hg.mozilla.org/projects/ionmonkey>

available in Richards et al. (2011)’s repository. These programs have been extracted from actual webpages, such as Google or Facebook, and provide a more faithful picture of JavaScript applications that we are likely to find in the wild. Even though it is hard to measure run time in these benchmarks with high confidence, we can use them to measure code size reduction and number of recompilations very reliably.

4.2 Evaluation

In this section we present the evaluation of the Speculative Value Specialization proposed in this work. We show the run time impact on the benchmarks considered and analyze the size of generated code, the overhead in compilation, and the impact on the number of compilations.

4.2.1 Run time impact

Tables 4.1 and 4.2 show the impact of our specialization in the run time on the SunSpider benchmark. Tables 4.3 and 4.4 show the run time impact on the V8 benchmark. And finally, Tables 4.5 and 4.6 show the run time impact on the Kraken benchmark. Each benchmark was executed 100 times, to reduce the imprecision of this experiment. For each test we present the execution time and standard deviation for different combinations of the optimizations that we have implemented (Section 3.3). The execution time measured in a test includes interpretation, compilation and native execution. PARAMETERSPEC is the parameter specialization that we have described in Section 3.2.2, augmented with the automatic inlining of functions passed as parameters.

We have run constant propagation without parameter specialization, as we show in the third column of the speedup tables, observing a run time slowdown of 0.88% in SunSpider, 0.5% in V8 and 0.08% in Kraken. Without parameter specialization, constant propagation has little room to improve the code, as IonMonkey’s global value numbering already eliminates most of the constants in the scripts. On the other hand, the combination of parameter specialization, constant propagation and dead-code elimination has produced one of our best results. Some optimizations enable others. As an example, in `string-unpack-code`, loop inversion has improved the effectiveness of IonMonkey’s invariant code motion, yielding a 29% speedup. Our implementation of array bounds check elimination did not give us a substantial speedup in any benchmark. We are using a simple approach, i.e., we only eliminate checks from arrays indexed by induction variables. Moreover, the alias analysis that ensures the correctness of this optimization is currently implemented in IonMonkey in a very simple way.

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•			•
LOOPINVERSION					•	
DEADCODEELIM			•			•
BOUNDCHECKELIM						

– SunSpider 1.0 (<http://www.webkit.org/perf/sunspider/sunspider.html>) –

3d-cube	2 ± 0.2	-2 ± 0.1	0 ± 0.2	2 ± 0.2	2 ± 0.2	0 ± 0.2
3d-morph	3 ± 0.8	0 ± 0.8	2 ± 1.0	3 ± 0.8	3 ± 0.7	2 ± 0.8
3d-raytrace	-2 ± 0.2	-1 ± 0.2	-3 ± 0.2	-3 ± 0.2	-2 ± 0.2	-3 ± 0.2
access-binary-trees	3 ± 0.1	0 ± 0.2	3 ± 0.2	3 ± 0.2	3 ± 0.2	3 ± 0.1
access-fannkuch	13 ± 0.2	-2 ± 0.2	22 ± 0.2	12 ± 0.2	9 ± 0.2	24 ± 0.2
access-nbody	-11 ± 0.2	-1 ± 0.2	-12 ± 0.2	-11 ± 0.2	-11 ± 0.2	-12 ± 0.2
access-nsieve	33 ± 0.3	0 ± 0.2	37 ± 0.2	34 ± 0.2	33 ± 0.2	35 ± 0.2
bitops-3bit-bits-in-byte	3 ± 0.2	-2 ± 0.2	2 ± 0.2	3 ± 0.2	3 ± 0.2	3 ± 0.2
bitops-bits-in-byte	1 ± 0.4	-1 ± 0.4	7 ± 0.4	15 ± 0.4	3 ± 0.4	4 ± 2.9
bitops-bitwise-and	0 ± 0.3	0 ± 0.4	0 ± 0.3	0 ± 0.4	0 ± 0.3	0 ± 0.4
bitops-nsieve-bits	-5 ± 0.2	-2 ± 0.2	-2 ± 0.2	-1 ± 0.2	-5 ± 0.2	-1 ± 0.2
controlflow-recursive	11 ± 0.7	-1 ± 0.7	10 ± 0.7	11 ± 0.7	11 ± 0.7	11 ± 0.7
crypto-aes	4 ± 1.0	-2 ± 1.0	2 ± 1.0	3 ± 1.0	4 ± 1.0	2 ± 1.0
crypto-md5	6 ± 0.3	0 ± 0.2	6 ± 0.3	6 ± 0.3	6 ± 0.3	6 ± 0.3
crypto-sha1	4 ± 0.2	-2 ± 0.2	2 ± 0.2	3 ± 0.2	4 ± 0.2	1 ± 0.2
date-format-tofte	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2
date-format-xparb	1 ± 0.2	0 ± 0.2	0 ± 0.2	1 ± 0.2	0 ± 0.2	2 ± 0.2
math-cordic	-4 ± 0.5	-1 ± 0.5	-4 ± 0.6	-2 ± 0.6	-3 ± 0.6	-4 ± 0.6
math-partial-sums	1 ± 0.3	0 ± 0.2	0 ± 0.2	1 ± 0.2	1 ± 0.2	0 ± 0.2
math-spectral-norm	-3 ± 0.2	-2 ± 0.2	-5 ± 0.2	-4 ± 0.2	-3 ± 0.2	-6 ± 0.2
regexp-dna	-2 ± 0.2	0 ± 0.2	1 ± 0.2	-1 ± 0.2	1 ± 0.2	-2 ± 0.2
string-base64	0 ± 0.2	-1 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	1 ± 0.2
string-fasta	5 ± 0.2	-1 ± 0.2	3 ± 0.2	3 ± 0.2	2 ± 0.2	3 ± 0.2
string-tagcloud	-6 ± 0.2	0 ± 0.2	-6 ± 0.2	-6 ± 0.2	-7 ± 0.2	-7 ± 0.2
string-unpack-code	0 ± 0.2	-1 ± 0.2	1 ± 0.2	0 ± 0.2	29 ± 0.2	-2 ± 0.2
string-validate-input	-12 ± 0.5	-1 ± 0.2	-12 ± 0.4	-12 ± 0.4	-12 ± 0.4	-12 ± 0.6
Average	1.73	-0.88	2.08	2.31	2.73	1.85

Table 4.1: Run time speedup for the SunSpider benchmark. Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•	•	•	•
LOOPINVERSION				•		•
DEADCODEELIM					•	•
BOUNDCHECKELIM			•		•	•

– SunSpider 1.0 (<http://www.webkit.org/perf/sunspider/sunspider.html>) –

3d-cube	2 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.1
3d-morph	3 ± 0.8	2 ± 0.8	2 ± 0.8	2 ± 0.7	2 ± 0.7	2 ± 0.7
3d-raytrace	-2 ± 0.2	-3 ± 0.2	-3 ± 0.2	-3 ± 0.2	-2 ± 0.2	-3 ± 0.2
access-binary-trees	3 ± 0.1	3 ± 0.1	3 ± 0.3	3 ± 0.2	3 ± 0.2	3 ± 0.2
access-fannkuch	13 ± 0.2	24 ± 0.2	22 ± 0.2	17 ± 0.2	24 ± 0.2	19 ± 0.2
access-nbody	-11 ± 0.2	-12 ± 0.2	-12 ± 0.2	-12 ± 0.2	-12 ± 0.2	-13 ± 0.2
access-nsieve	33 ± 0.3	35 ± 0.2	37 ± 0.2	38 ± 0.2	36 ± 0.3	36 ± 0.3
bitops-3bit-bits-in-byte	3 ± 0.2	3 ± 0.2	3 ± 0.2	3 ± 0.2	3 ± 0.2	1 ± 0.2
bitops-bits-in-byte	1 ± 0.4	4 ± 2.9	5 ± 0.4	1 ± 0.5	5 ± 0.4	-16 ± 0.4
bitops-bitwise-and	0 ± 0.3	0 ± 0.4	0 ± 0.3	0 ± 0.3	0 ± 0.4	0 ± 0.3
bitops-nsieve-bits	-5 ± 0.2	-1 ± 0.2	-5 ± 0.2	2 ± 0.2	-6 ± 0.2	-1 ± 0.2
controlflow-recursive	11 ± 0.7	11 ± 0.7	9 ± 0.7	9 ± 0.7	10 ± 0.7	11 ± 0.7
crypto-aes	4 ± 1.0	2 ± 1.0	2 ± 1.0	2 ± 1.0	2 ± 1.0	1 ± 1.0
crypto-md5	6 ± 0.3	6 ± 0.3	6 ± 0.3	6 ± 0.3	6 ± 0.3	5 ± 0.4
crypto-sha1	4 ± 0.2	1 ± 0.2	2 ± 0.2	2 ± 0.2	1 ± 0.2	1 ± 0.2
date-format-tofte	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.6	0 ± 0.2
date-format-xparb	1 ± 0.2	2 ± 0.2	-1 ± 0.2	-2 ± 0.2	1 ± 0.2	0 ± 0.2
math-cordic	-4 ± 0.5	-4 ± 0.6	-3 ± 0.6	-3 ± 0.7	-1 ± 0.6	-4 ± 0.5
math-partial-sums	1 ± 0.3	0 ± 0.2	0 ± 0.3	0 ± 0.3	0 ± 0.2	0 ± 0.2
math-spectral-norm	-3 ± 0.2	-6 ± 0.2	-6 ± 0.2	-6 ± 0.2	-6 ± 0.2	-6 ± 0.2
regex-dna	-2 ± 0.2	-2 ± 0.2	-5 ± 0.2	-1 ± 0.2	-4 ± 0.2	-3 ± 0.2
string-base64	0 ± 0.2	1 ± 0.2	0 ± 0.2	0 ± 0.1	1 ± 0.2	1 ± 0.2
string-fasta	5 ± 0.2	3 ± 0.2	4 ± 0.2	0 ± 0.2	4 ± 0.4	0 ± 0.2
string-tagcloud	-6 ± 0.2	-7 ± 0.2	-6 ± 0.2	-7 ± 0.2	-6 ± 0.2	-6 ± 0.2
string-unpack-code	0 ± 0.2	-2 ± 0.2	1 ± 0.2	29 ± 0.3	-1 ± 0.2	29 ± 0.2
string-validate-input	-12 ± 0.5	-12 ± 0.6	-12 ± 0.5	-13 ± 0.4	-12 ± 0.5	-13 ± 0.3
Average	1.73	-0.88	1.65	2.58	1.85	1.69

Table 4.2: Run time speedup for the SunSpider benchmark (part 2). Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•			•
LOOPINVERSION					•	
DEADCODEELIM				•		•
BOUNDCHECKELIM						

– V8 version 6 (<http://v8.googlecode.com/svn/data/benchmarks/v6/run.html>) –

deltablue	5 ± 0.6	-1 ± 0.8	7 ± 0.6	5 ± 0.6	6 ± 0.6	5 ± 0.6
earley-boyer	5 ± 0.2	0 ± 0.2	5 ± 0.1	6 ± 0.2	5 ± 0.2	7 ± 0.1
raytrace	12 ± 0.2	-1 ± 0.2	11 ± 0.2	12 ± 0.2	12 ± 0.3	12 ± 0.2
regex	1 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	1 ± 0.2	1 ± 0.2
richards	2 ± 0.1	-1 ± 0.1	3 ± 0.1	4 ± 0.1	-11 ± 0.1	5 ± 0.1
splay	-1 ± 0.5	0 ± 0.6	-1 ± 0.6	-1 ± 0.5	-1 ± 0.5	-1 ± 0.5
Average	4.00	-0.50	4.17	4.33	2.00	4.83

Table 4.3: Run time speedup for the V8 version 6 benchmark. Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•	•	•	•
LOOPINVERSION				•		•
DEADCODEELIM					•	•
BOUNDCHECKELIM			•		•	•

– V8 version 6 (<http://v8.googlecode.com/svn/data/benchmarks/v6/run.html>) –

deltablue	5 ± 0.6	-1 ± 0.8	6 ± 0.6	6 ± 0.5	5 ± 0.5	4 ± 0.5
earley-boyer	5 ± 0.2	0 ± 0.2	5 ± 0.2	3 ± 0.2	7 ± 0.1	6 ± 0.2
raytrace	12 ± 0.2	-1 ± 0.2	11 ± 0.2	10 ± 0.3	11 ± 0.2	13 ± 0.2
regex	1 ± 0.2	0 ± 0.2	0 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2
richards	2 ± 0.1	-1 ± 0.1	3 ± 0.1	-11 ± 0.1	5 ± 0.1	-7 ± 0.1
splay	-1 ± 0.5	0 ± 0.6	-1 ± 0.5	-2 ± 0.5	-1 ± 0.5	-2 ± 0.6
Average	4.00	-0.50	4.00	1.17	4.67	2.50

Table 4.4: Run time speedup for the V8 version 6 benchmark (part 2). Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•			•
LOOPINVERSION					•	
DEADCODEELIM				•		•
BOUNDCHECKELIM						•

– Kraken 1.1 (http://krakenbenchmark.mozilla.org) –						
ai-astar	0 ± 0.3	0 ± 0.2	0 ± 0.3	0 ± 0.2	0 ± 0.3	0 ± 0.3
audio-beat-detection	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3
audio-dft	1 ± 0.5	0 ± 0.4	1 ± 0.5	1 ± 0.5	2 ± 0.5	1 ± 0.5
audio-fft	4 ± 1.1	-1 ± 0.4	3 ± 0.3	4 ± 1.1	4 ± 1.2	4 ± 1.2
audio-oscillator	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.2
imaging-darkroom	0 ± 1.1	0 ± 0.8	0 ± 1.1	0 ± 1.1	1 ± 0.9	0 ± 0.9
imaging-gaussian-blur	1 ± 2.0	0 ± 2.1	2 ± 4.3	2 ± 2.0	3 ± 2.1	1 ± 3.7
json-parse-financial	0 ± 0.9	0 ± 0.9	0 ± 0.8	0 ± 0.8	0 ± 0.9	0 ± 0.8
json-stringify-tinderbox	0 ± 0.4	0 ± 0.5	1 ± 0.4	0 ± 0.5	1 ± 0.4	0 ± 0.5
stanford-crypto-aes	2 ± 0.7	0 ± 0.7	2 ± 0.7	2 ± 0.7	3 ± 0.6	2 ± 0.6
stanford-crypto-pbkdf2	1 ± 0.5	0 ± 0.5	1 ± 0.5	1 ± 0.5	1 ± 0.5	1 ± 0.4
stanford-crypto-sha256	0 ± 0.4	0 ± 0.4	-1 ± 0.4	0 ± 0.4	0 ± 0.3	0 ± 0.4
Average	0.75	-0.08	0.75	0.83	1.25	0.75

Table 4.5: Run time speedup for the Kraken benchmark. Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

Thus, if there exists any store instruction in the script being compiled, the elimination of bounds check instructions is considered unsafe and is not performed. If we run all our optimizations together, we do not obtain the best speedups. This happens because these optimizations are not cumulative: DEADCODEELIM and BOUNDCHECKELIM may, for instance, eliminate the same code.

4.2.2 Size of Generated Code

One of the benefits of our optimizations is code size reduction, which results from the combination of parameter specialization with dead code elimination. Figures 4.1, 4.2 and 4.3 presents the size of the machine code generated for functions of our three benchmarks with and without our approach. Notice that, due to recompilations, the same function may be translated in different ways. In this analysis, we consider only the smallest version that each compilation mode generates for each function. On average, we are able to reduce the size of the functions in SunSpider by 16.72%. This reduction, for V8 and Kraken, is 18.84% and 15.94%.

PARAMETERSPEC	•		•	•	•	•
CONSTANTPROPG		•	•	•	•	•
LOOPINVERSION				•		•
DEADCODEELIM					•	•
BOUNDCHECKELIM			•		•	•

– Kraken 1.1 (<http://krakenbenchmark.mozilla.org>) –

ai-astar	0 ± 0.3	0 ± 0.2	0 ± 0.3	0 ± 0.2	0 ± 0.3	0 ± 0.3
audio-beat-detection	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3	0 ± 0.3
audio-dft	1 ± 0.5	0 ± 0.4	1 ± 0.5	0 ± 0.5	-2 ± 0.5	-1 ± 0.5
audio-fft	4 ± 1.1	-1 ± 0.4	3 ± 1.2	4 ± 0.4	4 ± 1.2	5 ± 0.5
audio-oscillator	0 ± 0.2	0 ± 0.2	0 ± 0.2	0 ± 0.3	0 ± 0.3	0 ± 0.2
imaging-darkroom	0 ± 1.1	0 ± 0.8	0 ± 1.1	0 ± 1.1	0 ± 0.8	0 ± 1.0
imaging-gaussian-blur	1 ± 2.0	0 ± 2.1	2 ± 2.1	1 ± 3.7	1 ± 4.4	1 ± 2.1
json-parse-financial	0 ± 0.9	0 ± 0.9	0 ± 0.7	0 ± 0.7	0 ± 0.7	0 ± 0.8
json-stringify-tinderbox	0 ± 0.4	0 ± 0.5	0 ± 0.4	1 ± 0.6	0 ± 0.5	0 ± 0.5
stanford-crypto-aes	2 ± 0.7	0 ± 0.7	2 ± 0.6	2 ± 0.6	2 ± 0.7	3 ± 0.7
stanford-crypto-pbkdf2	1 ± 0.5	0 ± 0.5	1 ± 0.5	1 ± 0.4	1 ± 0.5	1 ± 0.5
stanford-crypto-sha256	0 ± 0.4	0 ± 0.4	-1 ± 0.3	-1 ± 0.3	0 ± 0.3	0 ± 0.3
Average	0.75	-0.08	0.67	0.67	0.50	0.75

Table 4.6: Run time speedup for the Kraken benchmark (part2). Percentage of speedup for different combinations of: parameter specialization (Section 3.2.2), constant propagation (Section 3.3.1), loop inversion (Section 3.3.2), dead code elimination (Section 3.3.3) and array bounds check elimination (Section 3.3.4).

PARAMETERSPEC	•		•	•	•	•	•	•	•	•
CONSTANTPROPG		•	•			•	•	•	•	•
LOOPINVERSION					•			•		•
DEADCODEELIM				•		•			•	•
BOUNDCHECKELIM							•		•	•
facebook-chrome	10.4	0.0	10.6	10.4	10.1	13.3	10.6	10.2	13.3	12.9
google-firefox	5.3	0.0	5.8	5.3	5.3	11.4	5.8	5.8	11.4	11.4
twitter-webkit	2.4	-0.1	3.4	2.4	2.4	22.1	3.4	3.4	22.1	22.1
yahoo-firefox	6.7	-0.2	6.4	6.7	6.7	42.5	6.4	6.4	42.5	42.5
sunspider-1.0	4.9	0.0	5.9	5.0	4.1	6.4	5.9	4.9	6.4	5.4
v8-v6	5.3	0.0	5.3	5.4	5.0	8.0	5.3	5.0	8.0	7.8
kraken-1.1	6.2	0.0	6.4	6.2	6.1	6.9	6.4	6.4	6.9	6.9

Table 4.7: Code Size Reduction, represented by posite values. Negative values represents an increase in the code size.

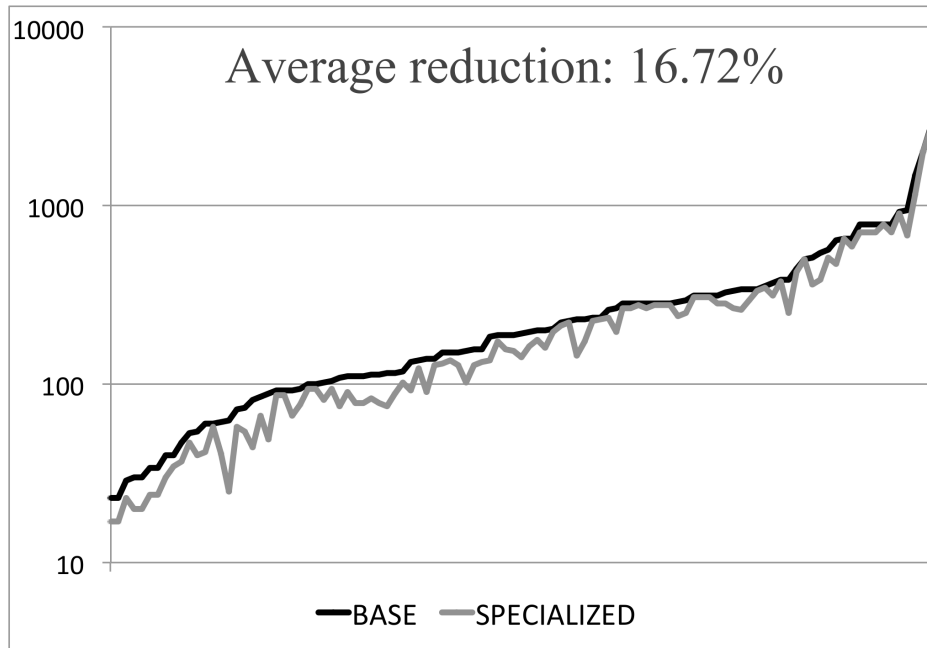


Figure 4.1: Size of generated code (log scale), in number of x86 assembly instructions per function, for SunSpider 1.0. Each point in the X axis is a function in our test suite. Functions are ordered by the size of the code that IonMonkey produces without our optimizations.

These numbers translate to real-world applications, as we also show in Table 4.7. We have run our techniques on the JavaScript benchmark automatically built from actual webpages, using Richards et al. (2011)’s tool, obtaining a code-size reduction of 13.3% for `www.facebook.com`, 11.4% for `www.google.com`, 22.1% for `www.twitter.com`, and 42.5% for `www.yahoo.com`. Notice that the JavaScript benchmarks that Richards *et al.* extract from an webpage depend on the browser used to visit that webpage. Thus, Table 4.7 shows, alongside each benchmark, the browser used to produce it. As expected, Table 4.7 shows that dead-code elimination is essential for code size reduction. Nevertheless, even without it, parameter specialization is still able to reduce the size of native binaries. Reduction, in this case, is due to the elimination of loads and stores necessary to map variables to memory, and the folding of type guards.

4.2.3 Compilation Overhead

Table 4.8 shows the impact of our optimizations in IonMonkey’s compilation time, i.e., the time that this engine spends analyzing, optimizing and generating code. Surprisingly, many of our configurations improve IonMonkey’s compilation time. As we

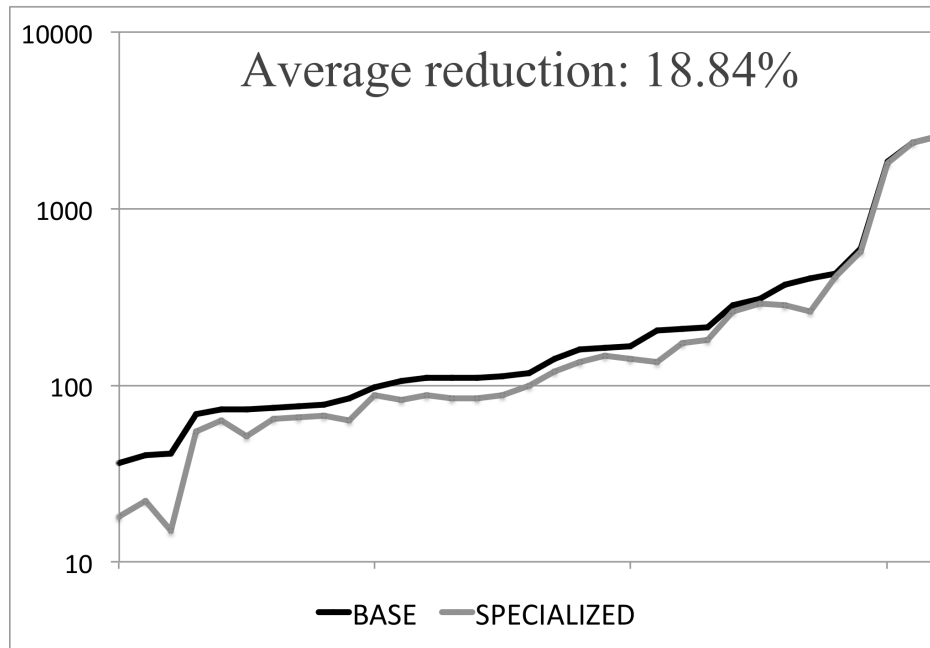


Figure 4.2: Size of generated code (log scale), in number of x86 assembly instructions per function, for V8 version 6. Each point in the X axis is a function in our test suite. Functions are ordered by the size of the code that IonMonkey produces without our optimizations.

have seen in Figures 4.1, 4.2 and 4.3, our optimizations decrease function sizes; thus, reducing the amount of work performed by the other phases of IonMonkey. Our key optimization, parameter specialization, has no overhead by construction. Instead of generating instructions that manipulate memory locations, we do it for constants. Additionally, this optimization improves the time of the register allocator, given that it reduces register pressure substantially.

4.2.4 Recompilations

A function will be recompiled by the JIT engine if an assumption made by the compiler stops being true or more information about the program becomes available. Recompilations are expensive, because they stop the execution of a function to generate a new version of it. In IonMonkey’s specific case, recompilations happen, for instance, to update type information or to perform function inlining. Since the value of arguments does not change until at least the function is invoked again, we perform our parameter based specialization even if a function is recompiled. Our approach may increase the number of recompilations of a function, because, in our case, each function will have

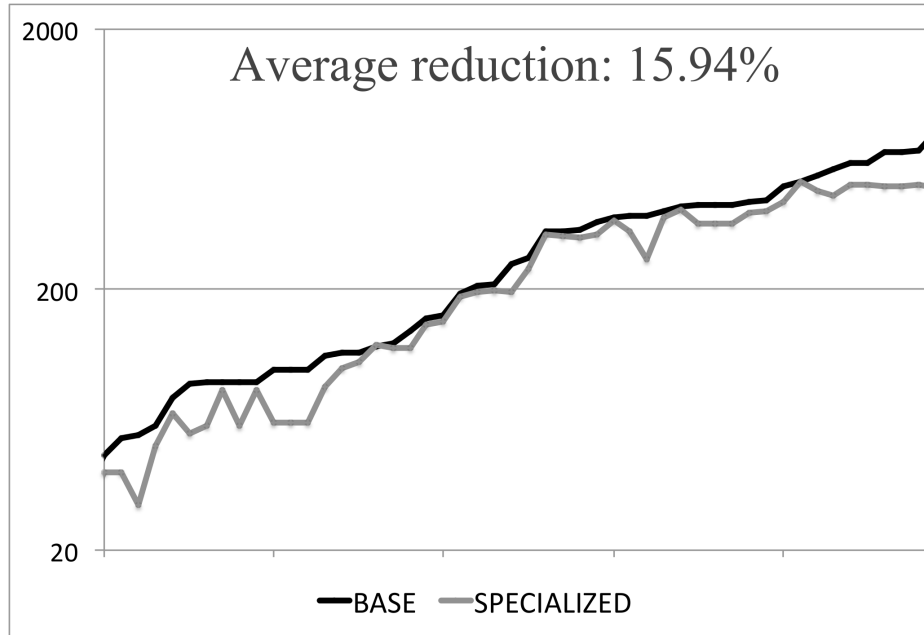


Figure 4.3: Size of generated code (log scale), in number of x86 assembly instructions per function, for Kraken 1.1. Each point in the X axis is a function in our test suite. Functions are ordered by the size of the code that IonMonkey produces without our optimizations.

PARAMETERSPEC	•		•	•	•	•	•	•	•	•
CONSTANTPROPG		•	•			•	•	•	•	•
LOOPINVERSION					•			•		•
DEADCODEELIM				•		•			•	•
BOUNDCHECKELIM							•		•	•

– (a) Compilation overhead (% arithmetic mean) –

Sunspider 1.0	-7.2	3.6	-4.3	-6.5	-7.3	-4.0	-3.8	-4.5	-3.6	-3.9
V8 version 6	1.5	2.1	4.0	2.6	1.7	3.6	4.3	3.4	3.8	3.4
Kraken 1.1	-3.0	3.0	-0.9	-2.4	-0.7	16.2	-0.5	1.6	16.5	1.4

– (b) Compilation overhead (% geometric mean) –

Sunspider 1.0	-8.7	3.6	-5.8	-8.0	-8.7	-5.5	-5.4	-6.0	-5.1	-5.4
V8 version v6	1.4	2.1	3.9	2.4	1.6	3.5	4.2	3.2	3.7	3.2
Kraken 1.1	-5.1	3.0	-2.9	-4.4	-2.9	5.2	-2.5	-0.5	5.6	-0.5

Table 4.8: Compilation overhead of different setups of our specialization engine. Numbers include our extra recom compilations. (a) Speedup relative to the baseline implementation (% arithmetic mean). (b) Speedup relative to the baseline implementation (% geometric mean). The baseline implementation, in all cases, does not do any value specialization.

PARAMETERSPEC	•		•	•	•	•	•	•	•	•
CONSTANTPROPG		•	•			•	•	•	•	•
LOOPINVERSION					•			•		•
DEADCODEELIM				•		•			•	•
BOUNDCHECKELIM							•		•	•
facebook-chrome	4.9	0	4.9	4.9	4.9	4.9	4.9	4.9	4.9	4.9
google-firefox	5.0	0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
twitter-webkit	23.1	0	23.1	23.1	23.1	23.1	23.1	23.1	23.1	23.1
yahoo-firefox	10.0	0	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
sunspider-1.0	3.1	0	3.6	3.1	1.3	3.1	3.6	1.8	3.1	1.3
v8-v6	6.3	0	5.8	6.1	5.4	6.1	6.3	5.4	6.1	5.1
kraken-1.1	3.4	0	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4

Table 4.9: Percentage of additional recompilations

to be recompiled whenever its is called for a second time with different arguments.

We have analyzed how often code is recompiled in our three benchmark suites with and without our run-time value specialization. For SunSpider, the number of compilations of the same function grows by 3.1% when using parameter specialization. This number is 6.3% for V8 and 3.4% for Kraken. Notice that these numbers consider the application of parameter specialization alone. The impact of the other combinations of optimizations can be seen in Table 4.9. These numbers vary, depending on which suite of optimizations we use, because they produce different code layouts, and the number of guarded operations in these different programs is not necessarily the same. Nevertheless, our approach tends to increase the number of recompilations by a small factor. Therefore, despite the highly speculative nature of this approach, its drawback, at least in our benchmarks, is not so big as one could at first expect.

4.2.5 Partial Specialization

A second invocation of a function may change some of these function’s parameters, but not all of them. Based on this observation, we have implemented a partial implementation policy, which tries to specialize a function one, and only one, time more. Our run-time environment, upon first seeing a call $f(a_0, b_0, c_0)$, generates a specialized version of f to the tuple (a_0, b_0, c_0) . Lets call this version f_{abc} . If we observe a second call of f , this time with the parameter set $f(a_0, b_0, c_1)$, we discard the original specialized function, but, instead of falling back to IonMonkey’s traditional code generation, we generate a function that is tailored to the tuple (a_0, b_0) . Lets call this function f_{ab} . Further calls of f will use f_{ab} , as long as the first two arguments remain the same. This

Benchmark	Comp	Spec	Partially	Deop
facebook-chrome	36	7	0	3
google-firefox	50	12	2	4
twitter-webkit	8	5	1	3
yahoo-firefox	3	1	1	1
SunSpider-1.0	124	56	20	30
V8 version 6	201	41	8	24
Kraken 1.1	74	38	6	21

Table 4.10: (Comp) Number of functions that have been compiled for each benchmark, (Spec) Number of functions that our heuristics decided to specialize, (Partially) Number of functions recompiled with partial specialization, (Deop) Number of deoptimized functions.

leaves the third argument free to change between calls, as it has been compiled generically. Thus, a call to $f(a_0, b_0, c_2)$ will still use f_{ab} , even if $c_1 \neq c_2$. On the other hand, if we observe a call such as $f(a_0, b_1, c_1)$, where $b_0 \neq b_1$, then we discard the customized code of f , and stop specialization. Table 4.10 shows the proportion of functions that we can specialize partially. Unfortunately, this heuristics did not produce results better than our first implementation, which only specializes once. The speedups that we produce for SunSpider, V8 and Kraken, using only parameter specialization are 1.63%, 2.33% and 1.72% respectively. Thus, in this case, we have observed gains only in Kraken, compared to the results seen in Table 4.1. If we use parameter specialization plus constant propagation, then our speedups are 2.73%, 1.5%, and 1.18%. In this case, we have observed gains in Sunspider and Kraken, at the expenses of a substantial loss in V8.

4.2.6 Specialization policy

We cache the arguments of each specialized function. In this way, if a function is called in sequence with the same arguments, then we reuse the specialized code. We distinguish *successfully specialized* and *deoptimized* functions. The former category represents the functions that are always called, throughout the entire program execution, with the same arguments. In this case, we have a win-win condition: we have produced more efficient code, and did not have to discard it later. If a function is called again with different arguments, then we discard its specialized code, and recompile it, this time producing generic code for it. Table 4.10 shows the proportion of functions that have been deoptimized because they were called with a different set of arguments.

On average, we specialize about 30-50% of the functions that IonMonkey compiles. This is the proportion of functions compiled due to long execution, instead of due to a large number of invocations. About 50-60% of these functions had to be deoptimized because they were called a second time with different parameters. Notice that these numbers are on par with the proportion of functions that are called with the same parameters, as pointed by Figure 3.3. We could partially specialize about 30-70% of these functions, given that only part of their arguments have changed.

4.3 Discussion

In this chapter we have discussed an empirical evaluation of our idea: Parameter-based Speculative Value Specialization. Considering the run-time impact of our approach, we have been able to speedup our benchmarks for all combinations of the different optimizations that we have tested. The highest speedup was achieved in V8 (4.83%) and the lower in Kraken (0.5%). As expected, the code specialization reduced the size of generated code for all benchmarks, including those produced by Richards et al. (2011)'s tool built from real web pages, like Facebook and Google. The compilation overhead was negative in many cases, a fact that we explain due to the reduction in code size that comes out of our optimizations. However, as this optimization is speculative, mispredictions may occur. Every misprediction causes a function recompilation. Although the number of recompilations can be small (6.1% in V8) for traditional benchmarks, in real web pages it can achieve 23.1% (twitter-webkit). The parameter specialization, by construction, has zero overhead. However, the cache mechanism used to reuse the specialized function force us to check all the arguments of a given function, every time this function is called. This run-time checks can represent an overhead that severely impacts the run time of a program. This case happens whenever a function calls a specialized function inside a loop, an event that often takes place in the Kraken benchmarks. Therefore, in spite of the exciting results seen in the Section 3.1, there exist still room for improvements in our implementation.

Chapter 5

Final Remarks

5.1 Future Work

While conducting the research summarized in this dissertation, we have identified several opportunities that can be explored in future work. One of them is to re-implement other classic compiler optimizations such as loop-unrolling and overflow-check elimination in the context of parameter-based value specialization. Another opportunity is to experiment our approach with different heuristics of compilation and cache. For instance, we specialize all functions that IonMonkey compiles. However, some benefit analysis could select only functions that will improve run time performance. We also cache only one binary per function. Thus, we can specialize only one different parameter set for the same function. We believe that this approach is the best trade-off, given the behavior of the JavaScript programs that we have found; however, more experiments are necessary to confirm this hypothesis.

5.2 Conclusion

The goal of this dissertation was to show that Speculative Value Specialization can improve JavaScript performance without the help of a profiler component, whose overhead could be prohibitive in certain environments. This work has described such speculative approach as well as a suite of value-based compiler optimizations that we have used to improve the execution time of an industrial-strength just-in-time JavaScript compiler. We believe that this work has pushed substantially further the notion of JIT speculation, as we produce highly-specialized binaries given the input values passed to a function. Our approach is most profitable when applied on functions that are called

always with the same parameters. We have demonstrated that these calls are very frequent, be it in well-known benchmarks, be it in actual web pages. We hope that the code optimization approach that we advocate in this work will open new directions to compiler research.

Software: our code, plus all the data that we have used in this work is available at our repository: <http://code.google.com/p/jit-value-specialization>.

Bibliography

- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Alexa Internet Inc (2010). Alexa top 500 global sites. <http://www.alexa.com/topsites>. Last access: november, 2012.
- Almási, G. and Padua, D. (2002). Majic: compiling matlab for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02*, pages 294--303, New York, NY, USA. ACM.
- Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88*, pages 1--11, New York, NY, USA. ACM.
- Alves, P. R. O., de Assis Costa, I. R., Pereira, F. M. Q., and Figueiredo, E. L. (2012). Parameter based constant propagation. In *Simposio Brasileiro de Linguagens de Programacao*. Sociedade Brasileira de Computacao.
- Anderson, C., Giannini, P., and Drossopoulou, S. (2005). Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05*, pages 428--452, Berlin, Heidelberg. Springer-Verlag.
- Apple Inc. (2012a). Sunspider javascript benchmark. <http://www.webkit.org/perf/sunspider/sunspider.html>. Last access: november, 2012.
- Apple Inc. (2012b). The webkit open source project. <https://www.webkit.org/>. Last access: november, 2012.
- Auslander, J., Philipose, M., Chambers, C., Eggers, S. J., and Bershad, B. N. (1996). Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996*

- Conference on Programming language design and implementation*, PLDI '96, pages 149--159, New York, NY, USA. ACM.
- Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97--113.
- Bala, V., Duesterwald, E., and Banerjia, S. (1999). Transparent dynamic optimization: The design and implementation of dynamo. Technical report, Technical Report HPL-1999-78, Hewlett-Packard Laboratories.
- Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, PLDI '00, pages 1--12, New York, NY, USA. ACM.
- Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. (2010a). SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM International Conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 708--725, New York, NY, USA. ACM.
- Bebenita, M., Chang, M., Wagner, G., Gal, A., Wimmer, C., and Franz, M. (2010b). 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. ACM.
- Bodík, R., Gupta, R., and Sarkar, V. (2000). Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, PLDI '00, pages 321--333, New York, NY, USA. ACM.
- Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. (2009). 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*, ICPOOLPS '09, pages 18--25, New York, NY, USA. ACM.
- Bolz, C. F., Leuschel, M., and Rigo, A. (2010). Towards just-in-time partial evaluation of prolog. In *Proceedings of the 19th International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'09, pages 158--172, Berlin, Heidelberg. Springer-Verlag.
- Calder, B., Feller, P., and Eustace, A. (1997). Value profiling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 259--269, Washington, DC, USA. IEEE Computer Society.

- Calder, B., Feller, P., Eustace, A., et al. (1999). Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1(1):1--6.
- Canal, R., González, A., and Smith, J. E. (2004). Software-controlled operand-gating. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 125--, Washington, DC, USA. IEEE Computer Society.
- Chambers, C. and Ungar, D. (1989). Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 146--160, New York, NY, USA. ACM.
- Chang, M., Mathiske, B., Smith, E., Chaudhuri, A., Gal, A., Bebenita, M., Wimmer, C., and Franz, M. (2011). The impact of optional type information on JIT compilation of dynamically typed languages. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 13--24, New York, NY, USA. ACM.
- Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., and Franz, M. (2009). Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual execution environments*, VEE '09, pages 71--80, New York, NY, USA. ACM.
- Cheng, B. and Buzbee, B. (2010). A jit compiler for android's dalvik vm. In *Google I/O Developer Conference*.
- Chevalier-Boisvert, M., Hendren, L. J., and Verbrugge, C. (2010). Optimizing Matlab through Just-In-Time Specialization. In *Proceedings of the 19th International Conference on Compiler Construction*, pages 46--65.
- Consel, C. and Noël, F. (1996). A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 145--156, New York, NY, USA. ACM.
- Costa, I., Alves, P., Santos, H. N., and Pereira, F. M. Q. (2013). Just-in-time value specialization. In *Proceedings of the 11th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, Washington, DC, USA. IEEE Computer Society.

- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451--490.
- Deutsch, L. P. and Schiffman, A. M. (1984). 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. ACM.
- Duesterwald, E. (2005). Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93(2):436--448.
- Eich, B. (2011). New javascript engine module owner. <https://brendaneich.com/2011/06/new-javascript-engine-module-owner/>. Last access: november, 2012.
- Engler, D. R. and Proebsting, T. A. (1994). Dcg: an efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 263--272, New York, NY, USA. ACM.
- Gabbay, F. and Mendelson, A. (1996). *Speculative execution based on value prediction*. Technion-IIT, Department of Electrical Engineering.
- Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghghat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. (2009). Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming language design and implementation*, PLDI '09, pages 465--478, New York, NY, USA. ACM.
- Gal, A., Probst, C. W., and Franz, M. (2006). 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. ACM.
- Garen, G. (2008). Surfin' safari: Announcing squirrelfish. <https://www.webkit.org/blog/189/announcing-squirrelfish/>. Last access: november, 2012.
- Godwin-Jones, R. (2010). Emerging technologies: New developments in WEB browsing and authoring. *Language Learning and Technology*, 14:9--15.

- Google Inc. (2012). V8 benchmark suite - version 6. <http://v8.googlecode.com/svn/data/benchmarks/v6/run.html>. Last access: november, 2012.
- Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. (2000). Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147--199.
- Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J. (1999). An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming language design and implementation, PLDI '99*, pages 293--304, New York, NY, USA. ACM.
- Guo, S.-y. and Palsberg, J. (2011). The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11*, pages 563--574, New York, NY, USA. ACM.
- Ha, J., Haghghat, M., Cong, S., and McKinley, K. (2009). A concurrent trace-based just-in-time compiler for single-threaded javascript. *PESPMA 2009*, page 47.
- Hackett, B. and Guo, S.-y. (2012). 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. ACM.
- Hölzle, U., Chambers, C., and Ungar, D. (1991). 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. Springer-Verlag.
- Inoue, H., Hayashizaki, H., Wu, P., and Nakatani, T. (2011). A trace-based java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11*, pages 246--256, Washington, DC, USA. IEEE Computer Society.
- Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H., and Nakatani, T. (1999). Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99*, pages 119--128, New York, NY, USA. ACM.
- Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis, SAS '09*, pages 238--255, Berlin, Heidelberg. Springer-Verlag.

- Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- jQuery Foundation (2012). jquery: The write less, do more, javascript library. <http://jquery.com/>. Last access: november, 2012.
- Keppel, D., Eggers, S., and Henry, R. (1991). *A case for runtime code generation*. Department of Computer Science and Engineering, University of Washington.
- Keppel, D., Eggers, S. J., and Henry, R. R. (1993). Evaluating runtime-compiled value-specific optimizations. Technical report, University of Washington.
- Logozzo, F. and Fähndrich, M. (2008). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the 2008 ACM symposium on Applied computing, SAC '08*, pages 184--188, New York, NY, USA. ACM.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184--195.
- Mehrara, M. and Mahlke, S. (2011). Dynamically accelerating client-side web applications through decoupled execution. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11*, pages 74--84, Washington, DC, USA. IEEE Computer Society.
- Meijer, E. and Drayton, P. (2005). Static Typing Where Possible, Dynamic Typing When Needed. *Workshop on Revival of Dynamic Languages*.
- Microsoft Corp. (2009). Msdn: Jscript (ecmascript3). <http://msdn.microsoft.com/en-us/library/hbxc2t98%28v=vs.85%29.aspx>. Last access: november, 2012.
- Mozilla Foundation (2012a). Javascript: Tracemonkey. <https://wiki.mozilla.org/JavaScript:TraceMonkey>. Last access: november, 2012.
- Mozilla Foundation (2012b). Kraken javascript benchmark. <http://krakenbenchmark.mozilla.org>. Last access: november, 2012.
- Mozilla Foundation (2012c). Mozilla developer network: Rhino. <https://developer.mozilla.org/en-US/docs/Rhino>. Last access: november, 2012.
- Mozilla Foundation (2012d). Mozilla developer network: Spidermonkey. <https://developer.mozilla.org/en/docs/SpiderMonkey>. Last access: november, 2012.

- Mozilla Foundation (2012e). Mozilla wiki: Ionmonkey. <https://wiki.mozilla.org/IonMonkey>. Last access: november, 2012.
- Mozilla Foundation (2012f). Mozilla wiki: Jaegermonkey. <https://wiki.mozilla.org/JaegerMonkey>. Last access: november, 2012.
- Muth, R., Watterson, S. A., and Debray, S. K. (2000). Code specialization based on value profiles. In *Proceedings of the 7th International Symposium on Static Analysis, SAS '00*, pages 340--359, London, UK, UK. Springer-Verlag.
- Nethercote, N. (2012). Spidermonkey is on a diet. <http://blog.mozilla.org/nnethercote/2011/11/01/spidermonkey-is-on-a-diet/>. Last access: november, 2012.
- Niyogi, S. (2010). Ieblog: The new javascript engine in internet explorer 9. <http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx>. Last access: november, 2012.
- Noel, F., Hornof, L., Consel, C., and Lawall, J. L. (1998). Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages, ICCL '98*, pages 132--, Washington, DC, USA. IEEE Computer Society.
- Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23--30.
- Richards, G., Gal, A., Eich, B., and Vitek, J. (2011). Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object oriented programming systems languages and applications, OOPSLA '11*, pages 677--694, New York, NY, USA. ACM.
- Richards, G., Lebesne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming language design and implementation, PLDI '10*, pages 1--12, New York, NY, USA. ACM.
- Rigo, A. (2004). 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. ACM.

- Samadi, M., Hormati, A., Mehrara, M., Lee, J., and Mahlke, S. (2012). Adaptive input-aware compilation for graphics engines. In *PLDI*, pages 13--22. ACM.
- Shankar, A., Sastry, S. S., Bodík, R., and Smith, J. E. (2005). Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05*, pages 327--343, New York, NY, USA. ACM.
- Sol, R., Guillon, C., Pereira, F. M. Q. a., and Bigonha, M. A. S. (2011). Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th International Conference on Compiler construction: part of the joint European conferences on theory and practice of software, CC'11/ETAPS'11*, pages 2--21, Berlin, Heidelberg. Springer-Verlag.
- Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. (2005). Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732--785.
- Thiemann, P. (2005). Towards a type system for analyzing javascript programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems, ESOP'05*, pages 408--422, Berlin, Heidelberg. Springer-Verlag.
- Tian, K., Jiang, Y., Zhang, E. Z., and Shen, X. (2010). An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, pages 125--139. ACM.
- Tian, K., Zhang, E., and Shen, X. (2011). A step towards transparent integration of input-consciousness into dynamic program optimizations. In *OOPSLA*, pages 445--462. ACM.
- Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181--210.
- Yermolovich, A., Wimmer, C., and Franz, M. (2009). Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages, DLS '09*, pages 79--88, New York, NY, USA. ACM.
- Zaleski, M., Brown, A. D., and Stoodley, K. (2007). Yeti: a gradually extensible trace interpreter. In *Proceedings of the 3rd International Conference on Virtual execution environments, VEE '07*, pages 83--93, New York, NY, USA. ACM.

- Zhang, W., Calder, B., and Tullsen, D. M. (2005). An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 87--98, Washington, DC, USA. IEEE Computer Society.
- Zhang, W., Checkoway, S., Calder, B., and Tullsen, D. (2007). Dynamic code value specialization using the trace cache fill unit. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 10--16. IEEE.