Parameter Based Constant Propagation

Péricles Rafael Oliveira Alves, Igor Rafael de Assis Costa,
Fernando Magno Quintão Pereira and Eduardo Lage Figueiredo

Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil
{periclesrafael,igor,fernando,figueiredo}@dcc.ufmg.br

Abstract. JavaScript is nowadays the lingua franca of web browsers. This programming language is not only the main tool that developers have to implement the client side of web applications, but it is also the target of frameworks such as Google Web Toolkit. Given this importance, it is fundamental that JavaScript programs can be executed efficiently. Just-in-time (JIT) compilation is one of the keys to achieve this much necessary efficiency. An advantage that a JIT compiler has over a traditional compiler is the possibility to use runtime values to specialize the target code. In this paper we push JIT speculation to a new extreme: we have empirically observed that many JavaScript functions are called only once during a typical browser section. A natural way to capitalize on this observation is to specialize the code produced by a function to the particular values that are passed to this function as parameters. We have implemented this approach on IonMonkey, the newest JIT compiler used in the Mozilla Firefox browser. By coupling this type of parameter specialization with constant propagation, a classical compiler optimization, we have been able to experimentally observe speedups of up to 25% on well-known algorithms. These gains are even more remarkable because they have been obtained over a worldly known, industrial quality JavaScript runtime environment.

1 Introduction

Dynamically typed programming languages are today widespread in the computer science industry. Testimony of this fact is the ubiquity of PHP, Python and Ruby in the server side of web applications, and the dominance of JavaScript on its client side. This last programming language, JavaScript, today not only works as a tool that developers may use to code programs directly, but also fills the role of an assembly language for the Internet [13]. The Google Web Toolkit, for instance, allows programmers to develop applications in Java or Python, but translates these programs to a combination of JavaScript and HTML [5]. Given this importance, it is fundamental that dynamically typed languages, which are generally interpreted, can be executed efficiently, and the just-in-time (JIT) compilers seem to be a key player to achieve this much needed speed [2].

However, in spite of the undeniable importance of a language such as JavaScript, executing its programs efficiently is still a challenge even for a JIT

compiler. The combination of dynamic typing and late binding hides from the compiler core information that is necessary to generate good code. The type of the values manipulated by a JavaScript program is only known at runtime, and even then it might change during program execution. Moreover, having a very constrained time window to generate machine code, the JIT compiler many times gives up important optimizations that a traditional translator would probably use. Therefore, it comes as no surprise that the industry and the academic community are investing a huge amount of effort in the advance of JIT technology [11, 13]. Our intention in this paper is to contribute further in this effort.

In this paper we discuss a key observation, and a suite of ideas to capitalize on it. After instrumenting the Mozilla Firefox browser, we have empirically observed that almost half the JavaScript functions in the 100 most visited websites in the Alexa index [1] are only called once. This observation motivates us to specialize these functions to their arguments. Hence, we give to the JIT compiler an advantage that a traditional compilation system could never have: the knowledge of the values manipulated at runtime.

We propose a form of constant propagation that treats the arguments passed to the function to be compiled as constants. Such approach is feasible, because we can check the values of these parameters at runtime, during JIT compilation. If the target function is only called once, then we have a win-win condition: we can generate simpler and more effective code, without having to pay any penalty. On the other hand, if the function is called more than once, we must recompile it, this time using a traditional approach, which makes no assumptions about the function arguments.

1.1 Why parameter specialization matters

The main motivation to our work comes out of an observation: most of the JavaScript functions in typical webpages are called only once. To corroborate this statement, Figure 1 plots the percentage of JavaScript functions called N times, $1 \leq N \leq 50$. To obtain this plot we have used the same methodology adopted by Richard *et al.* [10]: we have instrumented the browser, and have used it to navigate through the 100 most visited pages according to the Alexa index. This company offers users a toolbar that, once added to their browsers, tracks data about browsing behavior. This data is then collected and used to rank millions of websites by visiting rate.

From Figure 1 we see that the number of times each JavaScript function is called during a typical browser section clearly obeys a power law. About 47% of the functions are called only once, and about 59% of the functions are called at most twice. Therefore, many functions will be given only one set of arguments. Nevertheless, a traditional just-in-time compiler generates code that is general enough to handle any possible combination of parameters that their types allow. In this paper, we propose the exact opposite: lets specialize functions to their arguments; hence, producing efficient code to the common case. Functions that

[1] `http://www.alexa.com/`

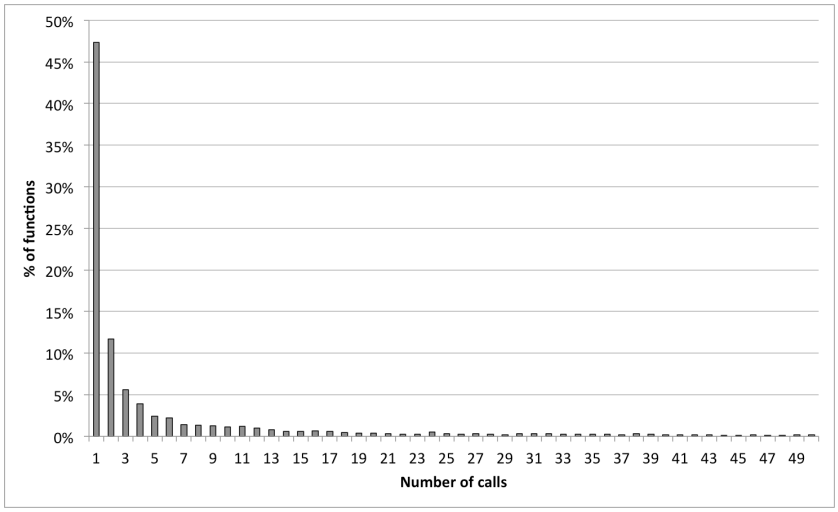

Fig. 1. A plot that shows how many times each different JavaScript function is called in a typical browser section in the 100 most visited pages according to the Alexa website.

are called more than once must either be re-compiled or interpreted. In this way we can produce super-specialized binaries, using knowledge that is only available at runtime; thus, achieving an advantage that is beyond the reach of any ordinary static compiler.

2 Parameter Based Method Specialization

In this section we illustrate our approach to runtime code optimization via an example. Then we explain how we can use runtime knowledge to improve constant propagation, a well-known compiler optimization.

2.1 Parameter Based Specialization by Example

We illustrate how a just-in-time compiler can benefit from parameter based specialization via the example program in Figure 2. The function `closest` finds, among the `n` elements of the array `v`, the one which has the smallest difference to an integer `q`. We see the control flow graph (CFG) of this program at the figure's right side. This CFG is given in the Static Single Assignment [9] (SSA) intermediate representation, which is adopted by IonMonkey, our baseline compiler. The SSA format has the core property that each variable name has only one definition site. Special instructions, called ϕ-functions, are used to merge together different definitions of a variable. This program representation simplifies substantially the optimizations that we describe in this paper.

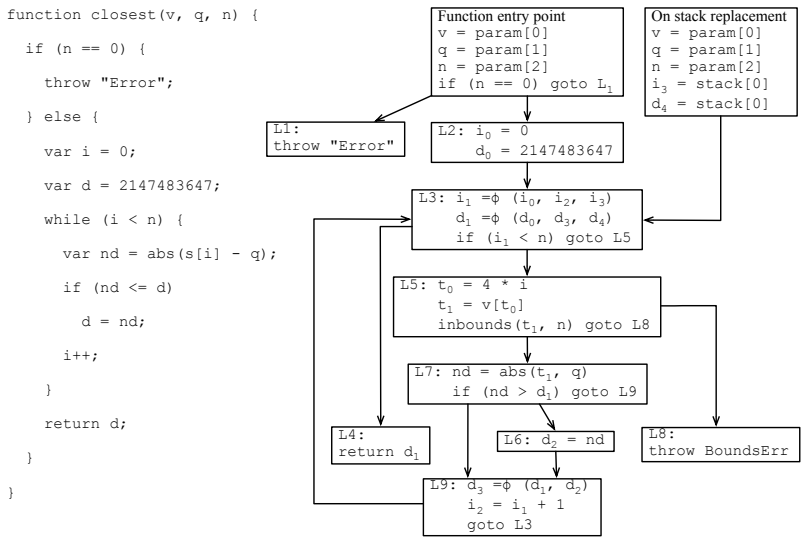

Fig. 2. (Left) The JavaScript program that will be our running example. (Right) A SSA-form, three-address code representation of the example.

The control flow graph in Figure 2(Right) differs from the CFG normally produced by a compiler because it has two entry points. The first, which we call *function entry point*, is the equivalent to the entry point of an ordinary CFG. The second, which we call the *On-Stack Replacement* block (OSR), is created by the just-in-time compiler, in case the function was compiled while being executed. All the functions that we optimize start execution from this block, as they are compiled only once. If a function is executed several times, then subsequent calls will start at the function entry.

The knowledge of the runtime values of the parameters improve some compiler optimizations. In this section we will show how this improvement applies onto four different compiler optimizations: dead-code elimination, array bounds check elimination, loop inversion and constant propagation. Figure 3(a) shows the code that we obtain after a round of dead-code elimination. Because we enter the function from the OSR block, the code that is reachable only from the function entry point is dead, and can be safely removed. This elimination also removes the test that checks if the array has a non zero size. Notice that even if reachable from the OSR block, we would be able to eliminate this test, given that we know that the result would be always positive.

Figure 3(b) shows the result of applying array bounds check elimination onto our example. JavaScript is a strongly typed language; thus, to guarantee the runtime consistency of programs, every array access is checked, so that memory is never indexed out of declared bounds. In our case, a simple combination of

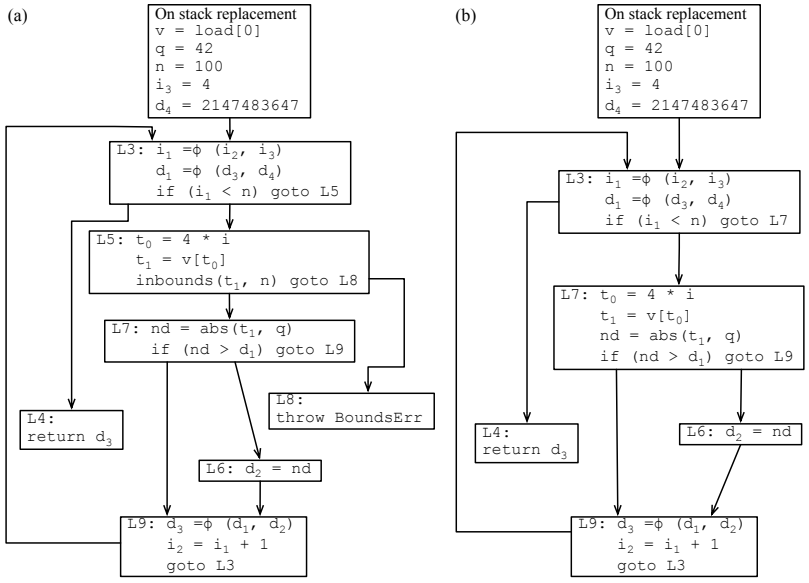

Fig. 3. The code that results from two different optimizations in sequence. (a) Dead-code elimination. (b) Array bounds check elimination.

range analysis [16], plus dead-code elimination is enough to remove the test performed over the limits of `v`. This limit, 100, is always greater than any value that the loop counter `i` can assume throughout program execution.

Figure 4(a) shows the result of applying loop inversion on the example, after dead-code elimination has pruned useless code. Loop inversion [15] converts a while into a do-while loop. The main benefit of this optimization is to replace the combination of conditional and unconditional branches used to implement the while loop by a single conditional jump, used to implement the repeat loop. Under ordinary circumstances an extra conditional test, wrapped around the while, is necessary, to certify that iterations will be performed only on non-null counters. However, given that we know that the loop will be executed at least once, this wrapping is not necessary.

Finally, Figure 4(b) shows the code that we obtain after performing constant propagation. Out of all the optimizations that we have discussed here, constant propagation is the one that most benefits from parameter specialization. Given that the parameters are all treated as constants, this optimization has many opportunities to transform the code. In our example, we have been able to propagate the array limit `n`, and the query distance `q`. Constant propagation is the optimization that we have chosen, in this paper, to demonstrate the effectiveness of parameter based code specialization. In the rest of this paper we will be discussing its implementation in our scenario, and its effectiveness.

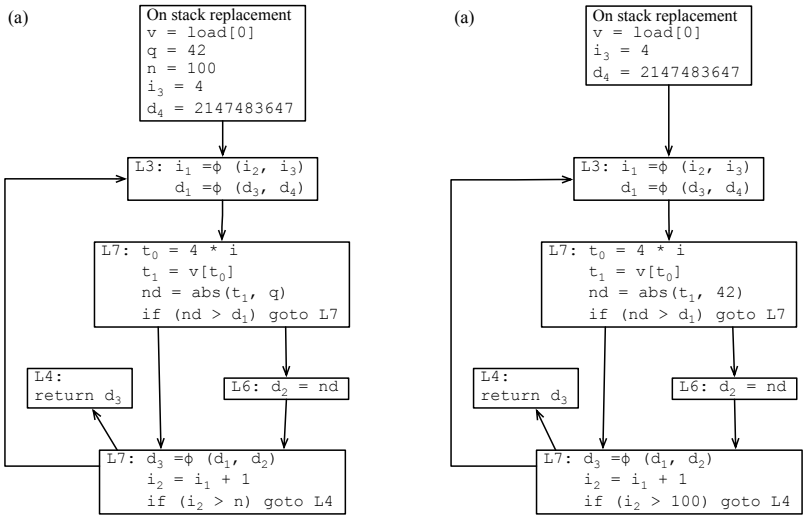

Fig. 4. Final code after two more optimizations. (a) Loop inversion. (b) Constant Propagation.

2.2 "Constification"

Argument based value specialization works by replacing the references to the parameters of a function about to be JIT compiled by the actual values of these parameters, in a process that we have dubbed *constification*. Before introducing the basic principles that underlie this technique, we will explain how the interpreter and the native code produced by the just-in-time compiler communicate. In this section we will describe the memory layout used by the SpiderMonkey interpreter; however, this organization is typical in other environments where the interplay between just-in-time compilation and interpretation happens, such as the Java Virtual Machine [2].

Whenever SpiderMonkey needs to interpret a JavaScript program, it allocates a memory space for this script, which, in Mozilla's jargon is called the *stack space*. This memory area will store the global data created by the script, plus the data allocated dynamically, for instance, due to function calls. A functions keep the data that it manipulates in a structure called activation record. This structure contains the function's parameters, its return address, the local variables, a pointer to the previous activation record, and a nesting link, which allows a nested function to find variables in the scope of the enclosing function. Activation records are piled on a stack, as different function calls take place. For instance, Figure 5(a) shows a stack configuration containing the activation records of two functions.

[2] See `Java SE HotSpot at a Glance`, available on-line

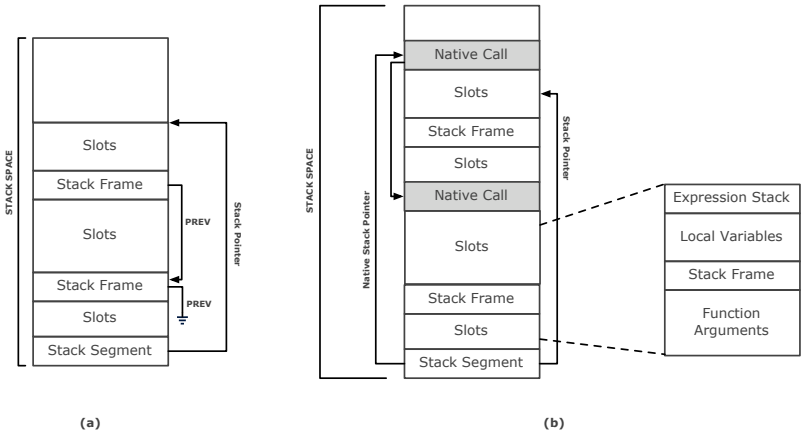

Fig. 5. SpiderMonkey's memory layout. (a) Interpretation. (b) Execution of native code. Data that is not visible to the interpreter is colored in gray. *Stack Segment* contains a pointer to the current top of stack, and to the chain of activation records of the native functions. Each activation record has a fixed-size area called the *stack frame*. *Slots* denote area whose layout depends on the function. Arguments of the function are stored before the stack frame, and the local variables are stored after it.

In principle, both the interpreter and the just-in-time compiled program could share the same stack of activation records, and thus we would have a seamless conversation between these two worlds. However, whereas the interpreter is a stack-based architecture, the native code runs on a register based machine. In other words, the two execution modes use different memory layouts. To circumvent this shortcoming, once an initially interpreted function is JIT compiled, its activation record is extended with a new memory area that is only visible to the JITed code. Figure 5(b) illustrates this new layout. In this example, we assume that both functions in execution in Figure 5(a) have been compiled to native code. The native code shares the activation records used by the interpreter – that is how it reads the values of the parameters, or writes back the return value that it produces upon termination. The interpreter, on the other hand, is oblivious to the execution of code produced by the just-in-time compiler.

Reading the values of parameters: JavaScript methods, in the Firefox browser, are initially interpreted. Once a method reaches a certain threshold of calls, or a loop reaches a certain threshold of iterations, the interpreter invokes the just-in-time compiler. At this point, a control flow graph is produced, with the two entry blocks that we have described in Section 2.1. Independent on the reason that has triggered just-in-time compilation, number of iterations or number of calls, the function's actual parameters are in the interpreter's stack, and can be easily retrieved. However, we are only interested in compilation due to

an excessive number of loop iterations. We do not specialize functions compiled due to an excessive number of calls; these functions are likely to be called many more times in the future. During the generation of the native code, we can find the values bound to the parameters of a function by inspecting its activation record. Reading these parameters has almost zero overhead when compared to the time to compile and execute the program.

After inspecting the parameter values, we redefine them in the two entry blocks of the CFG. For instance, in Figure 2(Right) we would replace the two load instructions, e.g., `v = param[0]` in the function entry and the OSR block, by the value of `v` at the moment the just-in-time compiler was invoked. Then, we replace all the uses of the parameters in the function body by their actual values. This last phase is trivial in the Static Single Assignment representation, because each variable name has only one definition site; hence, there is not the possibility of we wrongly changing a use that is not a parameter.

2.3 Argument Based Constant Propagation

In order to show that parameter based code specialization is effective and useful to just-in-time compilers, we have adapted the classic constant propagation algorithm [19] to make the most of the values passed to the functions as parameters. We call the ensuing algorithm *argument based constant propagation*, or ABCP for short. We have implemented ABCP in the IonMonkey [3] compiler. This compiler is the newest member of the *Monkey family*, a collection of JavaScript just-in-time engines developed in the Mozilla Foundation to be used in the Firefox browser. We chose to work in this compiler for two reasons. Firstly, because it is an open-source tool, which means that its code can be easily obtained and modified. Secondly, contrary to previous Mozilla compilers, IonMonkey has a clean design and a modern implementation. In particular, because it uses the SSA form, IonMonkey serves as a basis for the implementation of many modern compiler techniques.

Constant propagation, given its simple specification and straightforward implementation, is the canonical example of a compiler optimization [15, p.362]. Constant propagation is a *sparse* optimization. In other words, abstract states are associated directly with variables. The classic approach to constant propagation relies on an iterative algorithm. Initially all the variables are associated with the $\top$ abstract state. Then, those variables that are initialized with constants are added to a work list. If a variable is inserted into the worklist, then we know, as an invariant, that it has a constant value c_1, in which case its abstract state is c_1 itself. In the iterative phase, an arbitrary variable is removed from the worklist, and all its uses in the program code are replaced by the constant that it represents. It is possible that during this updating some instruction i is changed to use only constants in its right side. If such an event occurs, then the variable defined by i, if any, is associated to the value produced by i, and is inserted into the worklist. These iterations happen until the worklist is empty.

[3] `https://wiki.mozilla.org/IonMonkey`

At the end of the algorithm, each variable is known to have a constant value (C_i) or is not guaranteed to be a constant, and is thus bound to $\perp$.

Constant propagation suits well JIT compilers, because it is fast. The worst-case time complexity of this algorithm is $O(V^2)$, where V is the number of variables in the program. To derive this complexity, we notice that a variable can enter into the worklist at most once, when we find that it holds a constant value. A variable can be used in up to $O(I)$ program instructions, where I is the number of instructions. Normally $O(I) = O(V)$; thus, replacing a variable by the constant it represents takes $O(V)$ worst-case time. In practice a variable will be used in a few sites; therefore, constant propagation tends to be $O(V)$.

3 Experiments

In order to validate our approach, we have created a small benchmark that contains 8 well known algorithms, plus three programs from the SunSpider test suite. These benchmarks are publicly available at `http://code.google.com/p/im-abcp/source/browse/trunk/tests`. Figure 6 describes each of these programs.

Benchmark	LoC	Complexity	Description
SunSpider::math-cordic(R, C, A)	68	$O(R)$	Calls a sequence of transcendental functions R times.
SunSpider::3d-morph(L, X, Z)	56	$O(L \times X \times Z)$	Performs $L \times X \times Z$ calls to the sin transcendental function.
SunSpider::string-base64(T_{64}, B, T_2)	133	$O(\|T_{64}\|)$	Converts an array of integers to a Base-64 string.
matrix-multiplication(M_1, M_2, K, L, M)	46	$O(K \times L \times M)$	Multiplies a $K \times L$ matrix M_1 by a $L \times M$ matrix M_2.
k-nearest-neighbors(P, V, N, K)	47	$O(K \times N)$	Finds the K 2-D points stored in V that are closest of the 2-D point P.
rabin-karp(T, P)	46	$O(\|T\| \times \|P\|)$	Finds the first occurrence of the pattern P in the string —T—.
1d-trim(V, L, U, N)	22	$O(N)$	Given a vector V with N numbers, remove all those numbers that are outside the interval $[L, U]$.
closest-point(P, V, N)	35	$O(N)$	Finds, among the 2-D points stored in V, the one which is the closest to P.
tokenizer(S, P)	23	$O(\|S\| \times \|P\|)$	Splits the string S into substrings separated by the characters in the pattern P.
split-line(V, N, A, B)	41	$O(N)$	Separates the 2-D points stored in V into two groups, those below the line $y = Ax + b$, and those above.
string-contains-char(C, S)	13	$O(\|S\|)$	Tells if the string S contains the character C.

Fig. 6. Our benchmark suite. LoC: lines of JavaScript code.

IonMonkey does not provide a built-in implementation of Constant Propagation. Therefore, to demonstrate the effectiveness of our implementation, we compare it with the implementation of Global Value Numbering (GVN) already available in the IonMonkey toolset. GVN is another classic compiler optimization. IonMonkey uses the algorithm first described by Alpern *et al.* [1], which relies on the SSA form to be fast and precise. Alpern *et al.* have proposed two different approaches to GVN: pessimistic and optimistic. Both are available in IonMonkey. In this section we use the pessimistic approach, because it is considerably faster when applied to our benchmarks. All the implementations that we discuss in this section are intra-procedural. None of the IonMonkey built-in optimizations are inter-procedural, given the difficulties to see the entirety of

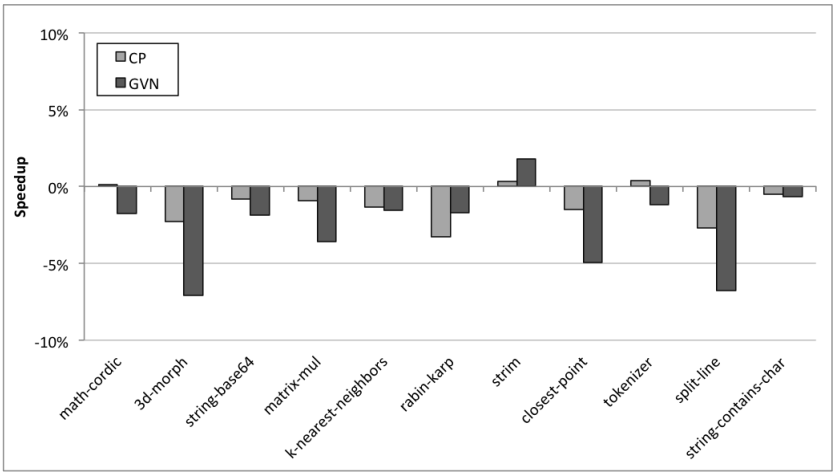

Fig. 7. Speedup of the original version of Constant Propagation and Global Value Numbering.

dynamically loaded programs. All the runtime numbers that we provide are the average of 1000 executions. We do not provide average errors, because they are negligible given this high quantity of executions.

Figure 7 compares our implementation of constant propagation with the implementation of global value number produced by the engineers that work in the Mozilla Foundation. The baseline of **all** charts in this section is the IonMonkey compiler running with no optimizations. The goal of figure 7 is to show that our implementation is not a straw-man: for our suite of benchmarks it produces better code than GVN, which is industrial-quality. We see in the figure that both optimizations slowdown the benchmarks. This slowdown happens because the total execution time includes the time to optimize the program, and the time that the program spends executing. Neither optimization, constant propagation or global value numbering, finds many opportunities to improve the target code in a way to pay for the optimization overhead. On the other hand, they all add an overhead on top of the just-in-time engine. However, the overhead imposed by constant propagation, a simpler optimization, is much smaller than the overhead imposed by global value numbering, as it is evident from the bars in Figure 7.

Figure 8 compares our implementation of constant propagation, with and without parameter specialization. We see that traditional constant propagation in fact slows down many of our benchmarks. Our implementation of the classic Rabin-Karp algorithm, for instance, suffers a 4% slowdown. Traditional constant propagation does not find many opportunities to remove instructions, given the very small number of constants in the program code, and given the fact that it runs intra-procedurally. On the other hand, the argument based implementation fares much better. It naturally gives us constants to propagate in all the bench-

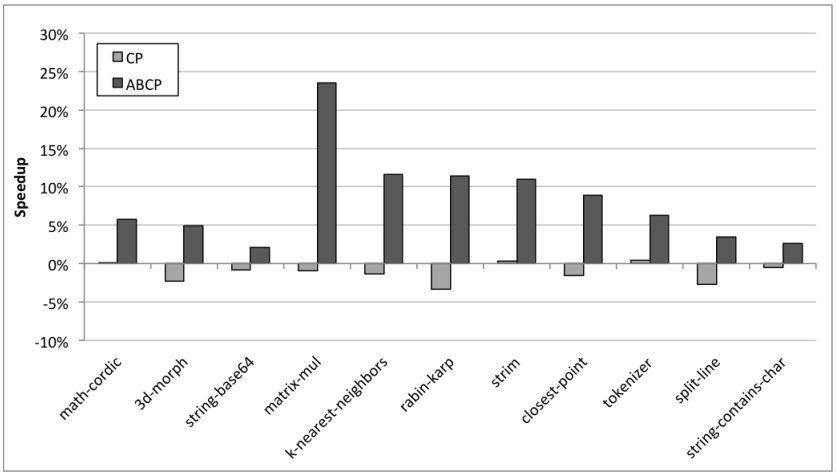

Fig. 8. Speedup of the original and parameter based version of Constant Propagation.

marks, and it also allows us to replace boxed values by constants. The algorithm of highest asymptotic complexity, matrix multiplication, experiments a speedup of almost 25%, for instance.

Figure 9 compares the implementation of global value numbering with and without parameter based specialization. Contrary to constant propagation, global value numbering does not benefit much from the presence of more constants in the program text. We see, for instance, that the SunSpider's `string-based64` benchmark suffers a slowdown of over 30%. This slowdown happens because of the time spent to load and propagate the values of the arguments. None of these arguments are used inside loops - although expressions derived from them are - and thus GVN cannot improve the quality of these loops. On the other hand, again we observe a speed up in matrix multiplication. This speedup does not come from GVN directly. Rather, it is due to the fact that our constification replaces the loop boundaries by integer values, as a result of the initial value propagation that we perform upon reading values from the interpreter stack. Figure 10 compares constant propagation and global value numbering when preceded by parameter based value specialization. On the average the argument based constant propagation delivers almost 25% more speedup than argument based global value numbering.

Figure 11 gives us some further subsidies to understand the speedups that parameter specialization delivers on top of constant propagation. First, we notice that in general constant propagation leads to less code recompilation. In general a just-in-time compiler might have to re-compile the same function several times, while this function is still executing. These recompilations happen because some assumptions made by the JIT may no longer hold during program execution, or it may infer new facts about the program. For instance, the JIT

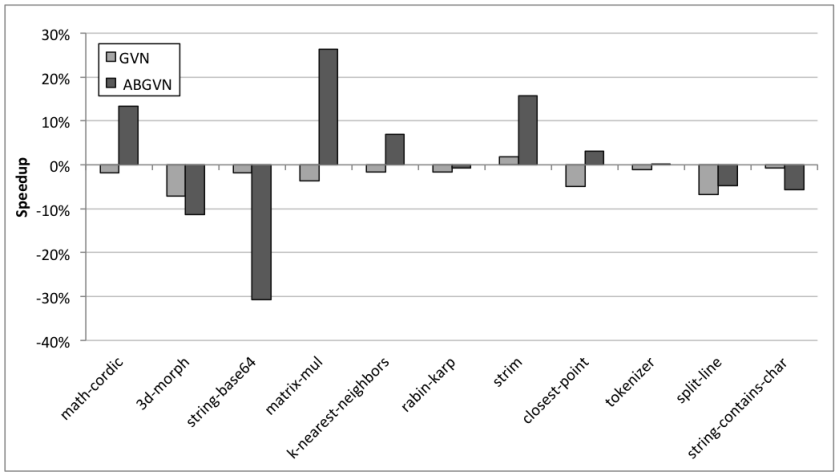

Fig. 9. Speedup of the original and parameter based version of Global Value Numbering.

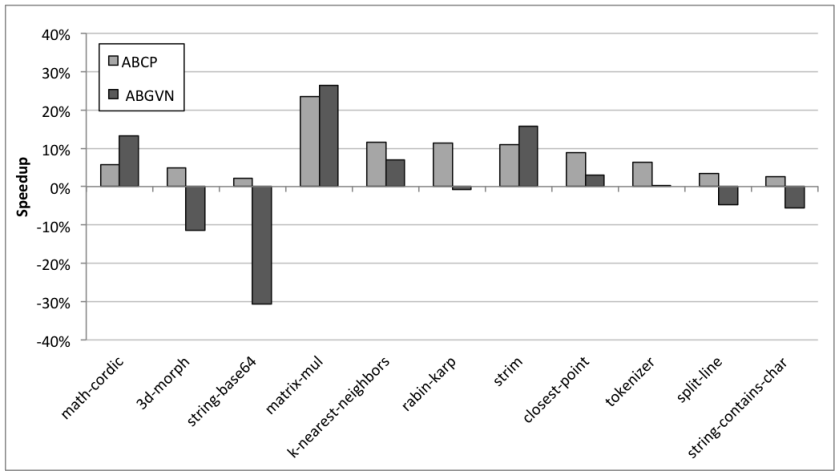

Fig. 10. Speedup of the parameter based versions of Constant Propagation and Global Value Numbering.

may discover that a reference is used as an integer inside a loop, and this new knowledge may trigger another compilation. If, eventually this reference receives a non-integer value, or some arithmetic operation causes this integer to overflow, then a new compilation is in order. Second, it is clear from the table that argument based specialization considerably improves the capacity of constant propagation to eliminate instructions. When an instruction is eliminated be-

	CP				ABCP			
Benchmark	R	I	F	% F	R	I	F	% F
math-cordic	1	287	0	0%	1	295	22	7%
3d-morph	2	582	0	0%	2	600	63	11%
string-base64	3	1503	30	2%	3	1519	58	4%
matrix-mul	8	1574	0	0%	3	558	84	15%
k-nearest-neighbors	2	530	4	1%	1	432	46	11%
rabin-karp	2	583	9	2%	2	595	57	10%
strim	1	154	0	0%	0	82	13	16%
closest-point	1	228	0	0%	0	142	13	9%
tokenizer	2	296	3	1%	2	308	39	13%
split-line	2	390	0	0%	1	300	26	9%
string-contains-char	0	58	0	0%	0	64	15	23%

Fig. 11. A comparison, in numbers, between constant propagation, and argument based constant propagation. R: number of recompilations. I: total number of instructions produced by the JIT compiler. F: number of instructions removed (folded) due to constant propagation. %F: percentage of folded instructions.

cause all the variables that it uses are constants, we say that the instruction has been *folded*. At least in our benchmark suite, traditional constant propagation does not find many opportunities to fold instructions. However, once we replace parameters by constants, it produces remarkably good results. In some cases, as in the function `string-contains-char`, it can eliminate almost one fourth of all the native instructions generated.

4 Related Work

The dispute for market share among Microsoft, Google, Mozilla and Apple has been known in recent years as the "browser war" [17]. Performance is a key factor in this competition. Given that the performance of a browser is strongly connected to its capacity to execute JavaScript efficiently, today we watch the development of increasingly more reliable and efficient JavaScript engines.

The first just-in-time compilers were method based [2]. This approach to just-in-time compilation is still used today with very good results. Google's V8 [4] and Mozilla's JaegerMonkey [5], are method-based JIT compilers. Method based compilation is popular for a number of reasons. It can be easily combined with many classical compiler optimization, such as Value Numbering and Loop Invariant Code Motion. It also capitalizes on decades of evolution of JIT technology, and can use old ideas such as Self's style type specialization [6]. Furthermore, this technique supports well profiling guided optimizations [8], such as Zhou's dynamic elimination of partial redundancies [22] and Bodik's array bounds checks elimination [4]. IonMonkey, the compiler that we have adopted as a baseline in

[4] `http://code.google.com/p/v8/`
[5] `https://wiki.mozilla.org/JaegerMonkey`

this paper, is an method-based JIT compiler, that resorts to hybrid type inference [14] in order to produce better native code.

A more recent, and substantially different JIT technique is trace compilation [3]. This approach only compiles linear sequences of code from hot loops, based on the assumption that programs spend most of their execution time in a few parts of a function. Trace compilation is used in compilers such as Tamarimtrace [7], HotpathVM [12], Yeti [21] and TraceMonkey [11]. There exist code optimization techniques tailored for trace compilation, such as Sol *et al.*'s [18] algorithm to eliminate redundant overflow tests. There are also theoretical works that describe the semantics of trace compilers [20].

5 Conclusion

In this paper we have introduced the notion of *parameter based code specialization*. We have shown how this technique can be used to speedup the code produced by IonMonkey, an industrial-strength just-in-time compiler that is scheduled to be used in the Mozilla Firefox browser. Parameter based specialization has some resemblance to partial evaluation; however, we do not pre-execute the code in order to improve it. On the contrary, we transform it using static compiler optimizations, such as constant propagation for instance. Our entire implementation plus the benchmarks that we have used in this paper are completely available at `http://code.google.com/p/im-abcp/`.

We believe that parameter based code specialization opens up many different ways to enhance just-in-time compilers. In this paper we have only scratched the tip of these possibilities, and much work is still left to be done. In particular, we would like to explore how different compiler optimizations fare in face of parameter specialization. From our preliminary experiments we know that some optimizations such as constant propagation do well in this new world; however, optimizations such as global value numbering cannot benefit much from it. We have already a small list of optimizations that we believe could benefit from our ideas. Promising candidates include loop inversion, loop unrolling and dead code elimination.

Even though we could add non-trivial speedups on top of Mozilla's JavaScript engine, our current implementation of parameter based code specialization is still research-quality. We are actively working to make it more robust. Priorities in our to-do list include to specialize functions that are compiled more than once, and to cache the values of the specialized parameters, so that future function calls can use this code. Nevertheless, the preliminary results seem to be encouraging.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM, 1988.
2. John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

3. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12. ACM, 2000.
4. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
5. Prabhakar Chaganti. *Google Web Toolkit GWT Java AJAX Programming*. PACKT, 1st edition, 2007.
6. Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.
7. Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: Trace compilation for the next generation web applications. In *VEE*, pages 71–80. ACM, 2009.
8. Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
10. Richards G., Lebresne S., Burg B., and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.
11. Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Jess Ruderman, Edwin Smith, Rick Reitmair, Mohammad R. Haghighat, Michael Bebenita, Mason Change, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM, 2009.
12. Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *VEE*, pages 144–153, 2006.
13. Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, pages 31–44. ACM, 2012.
14. Brian Hackett and Shu yu Guo. Fast and precise hybrid type inference for JavaScript. In *PLDI*. ACM, 2012.
15. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
16. Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.
17. Stephen Shankland. How JavaScript became a browser-war battleground. http://www2.galcit.caltech.edu/ jeshep/GraphicsBib/NatBib/node3.html, 2009. Accessed in 2012 (Apr 30).
18. Marcos Rodrigo Sol Souza, Christophe Guillon, Fernando Magno Quintao Pereira, and Mariza Andrade da Silva Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21, 2011.
19. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.
20. Shu yu Guo and Jens Palsberg. The essence of compiling with traces. In *POPL*, page to appear. ACM, 2011.
21. Mathew Zaleski. *YETI: A Gradually Extensible Trace Interpreter*. PhD thesis, University of Toronto, 2007.
22. Hucheng Zhou, Wenguang Chen, and Fred C. Chow. An SSA-based algorithm for optimal speculative code motion under an execution profile. In *PLDI*, pages 98–108. ACM, 2011.