Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler

Rafael Auler[1], Edson Borin[1], Peli de Halleux[2], Michał Moskal[2], and Nikolai Tillmann[2]

[1] University of Campinas, Brazil
{auler,edson}@ic.unicamp.br
[2] Microsoft Research, Redmond, WA, USA
{jhalleux,micmo,nikolait}@microsoft.com

Abstract. JavaScript has long outpaced its original target applications, being used not only for coding complex web clients, but also web servers, game development and even desktop applications. The most appealing advantage of moving applications to JavaScript is its capability to run the same code in a large number of different devices. It is not surprising that many compilers target JavaScript as an intermediate language. However, writing optimizations and analyses passes for a compiler that emits JavaScript is challenging: a long time spent in optimizing the code in a certain way can be excellent for some browsers, but a futile effort for others. For example, we show that applying JavaScript code optimizations in a tablet with Windows 8 and Internet Explorer 11 increased performance by, on average, 5 times, while running in a desktop with Windows 7 and Firefox decreased performance by 20%. Such a scenario demands a radical new solution for the traditional compiler optimization flow. This paper proposes collecting web clients performance data to build a crowdsourced compiler flag suggestion system in the cloud that helps the compiler perform the appropriate optimizations for each client platform. Since this information comes from crowdsourcing rather than manual investigations, fruitless or harmful optimizations are automatically discarded. Our approach is based on live measurements done while clients use the application on real platforms, proposing a new paradigm on how optimizations are tested.

Keywords: Adaptive compilation, JavaScript engines, just-in-time compilation

1 Introduction

JavaScript started as a simple non-professional scripting language in 1995 to support small-scale client-side logic in the earliest versions of the Netscape Navigator web browser. By now the language has become so pervasive that it invaded even non-web domains previously reserved for classic programming languages. With the availability of high performing virtual machines like Node.js [28] and efficient Just-in-Time (JIT) compilation technology, not only are complex web

applications moving its logic to client-side JavaScript, but server applications are also being coded in JavaScript as much of the server-side programming logic fits nicely with JavaScript closures. Overall, JavaScript's popularity made the language common for coding web clients, web servers, game development and even desktop applications [1].

The most appealing advantage of moving applications to JavaScript is its capability to run the same code in a large number of different devices. This was a major factor for the design of TouchDevelop [29], a modern, device independent browser-based programming language and development environment. TouchDevelop offers a platform for users to create scripts in its own custom language, designed for simplicity of programming on touch devices. As far as we are aware, TouchDevelop is currently the most advanced environment for programming on the phone. While the original purpose was to create simple scripts selecting coding structures with your finger, it turned out to be so easy to program that it began being adopted as a teaching environment in schools, by hobbyist programmers, and even by professional developers using their phone to program while on the go.

TouchDevelop scripts inherit the characteristics of their platform and run on the browser as JavaScript code, so there is a compiler that translates the TouchDevelop language to JavaScript. JavaScript is a hot target for compilers, as seen by the increasing number of projects that compile code to it, such as the Google Web Toolkit [2] by Google, TypeScript [7] by Microsoft, Dart [5] by Google or the Emscripten [30] used in the LLVM [10] community.

However, the ability to run on many different environments also brings new challenges when it comes to ensure good performance of the scripts. Since clients have different browsers to choose from and each browser implements its own JavaScript engine (e.g. SpiderMonkey [3], V8 [9], JavaScriptCore (aka. Nitro) [8] or Chakra [6]), optimizing the JavaScript code becomes a guessing game because each engine has its own optimizations and limitations.

Moreover, writing optimizations and analyses passes for a compiler that emits JavaScript is further complicated because of the time spent in optimizing code, which affects user experience when the compilation is not offline, as in TouchDevelop. The compiler can spend a significant amount of time to apply an optimization that is worthless for a particular JIT engine or even make the script slower. There is a number of possible causes: the underlying JIT engine may already apply this kind of optimization; changing the code in a particular way may preclude further JIT optimizations by the browser; or perhaps this particular issue was never the true performance bottleneck of this system. Overall, it is expensive to handle all particularities of each platform.

To overcome these problems, we developed a crowdsourced approach to drive our JavaScript compiler optimizations. We use a benchmark set of TouchDevelop scripts to exercise common performance bottlenecks and compile these scripts with different optimizations in different clients, storing the results of each client in the cloud. This enables us to characterize how each system responds to our optimizations and this information gets uploaded to the cloud. When another

user that uses the same platform compiles the TouchDevelop script to JavaScript, the system queries the cloud to know the best set of flags, or optimizations to apply, that best suits her system.

In this paper we describe this system in detail and report on our experience with our crowdsourced flag inference to circumvent JIT engines limitations. We also present a set of optimizations that addresses common language implementation issues when compiling to JavaScript that is able to speed applications up by 30x.

The main contributions of this work are as follows:

– We identify how JavaScript performance can vary from browser to browser and present three optimizations that handle the limitations of each JavaScript engine regarding common language implementation issues;

– We describe an approach to performance data crowdsourcing of web client software;

– We present a compiler flag suggestion system for a compiler that targets the JavaScript language;

– We implement and test these concepts in a real web-based programming environment used by tens of thousands of users, TouchDevelop, and present data from more than a thousand users that collaborated with the project.

This paper is organized as follows. Section 2 presents our benchmark selection, Section 3 discusses how performance data is reported to the cloud, Section 4 presents the overall structure of the TouchDevelop compiler, Section 5 presents the experimental results, Section 6 discusses related work and Section 7 presents the conclusions.

2 Selection of Benchmarks

The selection of benchmarks shapes the development of compiler optimizations and the performance bottlenecks identification. At the same time that it is at the crux of the performance study of any computer system [27], it is also impossible to build a set of programs that exercises the execution paths of all possible programs that can be written in a general purpose programming language.

To commit to a specific set of benchmarks is an important step, and therefore we chose the benchmarks from the Computer Languages Benchmarks Game (CLBG) [4] because of the benefit of comparing the TouchDevelop language performance with several other languages that had the same programs implemented using them. Table 1 presents the 8 chosen benchmark programs from the Computer Language Benchmarks Game.

Our benchmark selection includes all of the CLBG programs, except for those that use thread support, since TouchDevelop does not support multi-threading nor does the underlying language that TouchDevelop compiles to, JavaScript.

The CLBG website also publishes results and implementations of the same programs in optimized JavaScript. This enables us to compare the performance of the code generated by the TouchDevelop compiler against a manually written version of the same program in JavaScript.

Table 1. Description of the selected benchmark programs taken from the CLBG website [4].

Program	Description
n-body	Perform an N-body simulation of the Jovian planets
fannkuch-redux	Repeatedly access a tiny integer-sequence
fasta	Generate and write random DNA sequences
spectral-norm	Calculate an eigenvalue using the power method
reverse-complement	Read DNA sequences and write their reverse-complement
mandelbrot	Generate a Mandelbrot set and write a portable bitmap
k-nucleotide	Repeatedly update hashtables and k-nucleotide strings
binary-trees	Allocate and deallocate many binary trees

3 Live crowdsourced performance measurement

A primary issue in the live performance measurement of web client software, which is the measurement of the users experience while they are using the platform, is how to cope with the diversity of platforms where the measurements are taking place. Specifically, how to compare and keep track of the performance of the web client software if the computers that run it are constantly changing?

For example, a naïve comparison can mistakenly report code performance improvements between two measurements simply because the latest measurement took place in a client device that is more powerful than the device where previous measurements were taken. To tackle this issue, we first start by aggregating data by each different platform string taken from the *User Agent* string in HTTP requests. We tallied over 30 different client platforms that were using the TouchDevelop web client. This allows us to examine separately the behavior on each different kind of platform.

Table 2 shows the number of synchronization requests to update the web client with respect to the cloud data, a measurement of the activity by platform. Along with the data required to identify the platform, we also send to the server the wall time that this device took to run our benchmarks in JavaScript. For example, the first line shows the platform with the highest activity measured, a version of the Windows Phone 8 with the Internet Explorer 10 browser with 11,756 requests, whose average time to complete the execution of the JavaScript benchmark is 688.86 ms and the standard deviation is 266.84 ms. This data was extracted from a batch of 50,000 requests.

Categorizing the performance data with respect to the platform string is useful, but not enough. Table 2 shows that in a given platform, there is a very high standard deviation between all the measurements of the run time to complete the same task. While identifying devices by the *User Agent* string gives some characteristics of the client system, we are not able to fully identify underlying hardware configuration, which plays a crucial role in the final system speed and cause significant differences in the reported run time to complete the same task.

To allow us to study the performance improvement of web clients regardless of the client speed, we adopted the run time of the JavaScript version of the

Table 2. Frequency of use of the 5 most popular TouchDevelop web clients by platform string, in number of synchronization requests (total of 50,000 requests by 32 different platforms during August 2013).

Platform	Requests	Average Time to Complete Benchmark (ms)
Windows Phone 8.0.10211.0 with IE10	11,756	688.86 ± 266.84 ms
Windows 7 Desktop with Chrome	6,046	149.52 ± 172.47 ms
Windows 8 Desktop with Chrome	5,731	144.93 ± 140.82 ms
Windows Phone 8.0.10328.0 with IE10	3,998	623.18 ± 219.17 ms
Windows 8 Desktop with IE10	3,336	572.26 ± 1638.84 ms

programs featured in our subset of the Computer Language Benchmarks Game run on a particular small input, as a reference time for this platform, the *unit time*. It is an indication of the processing power of the platform, measured by the time it took to complete (the lower, the better).

Time measurements reported to the cloud comes with the unit time as well, along with the raw time required to complete a task. The raw time is divided by the reference time, and finally this ratio is reported as an approximated task performance score.

Figure 1 shows a diagram explaining how different devices report performance results to the cloud. The raw run time required to run a certain task, for example, a script execution, is divided by the unit time, a reference of its computational power.

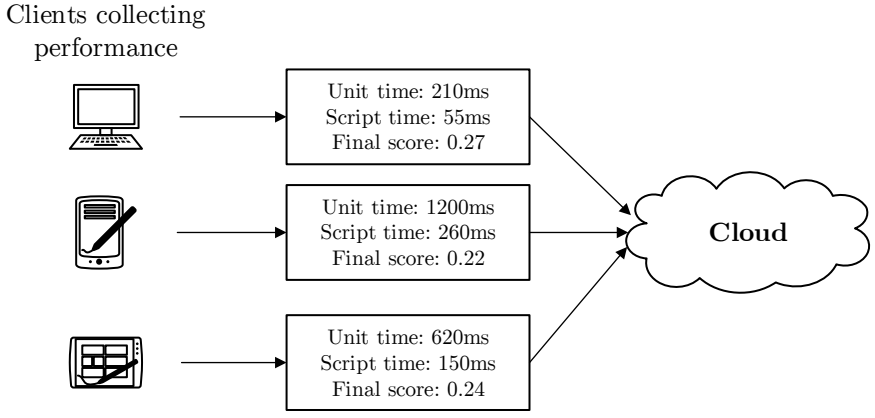

Fig. 1. A diagram showing how performance of different client devices is reported back to the cloud.

For example, a desktop with Mozilla Firefox 23 typically executes the *unit benchmark* in 210 ms, while a slower smartphone with Internet Explorer 10 in 1200 ms and an intermediary tablet device with Chrome in 600 ms. Suppose we want to test the execution speed of a script S. The script execution time is very different across these devices, but the run time of S divided by the unit time will be closer even among different devices, since the slowdown caused by the different device speed is factored out.

A special case is that of measuring our optimizations effects on the benchmark programs, as reported in this paper. The benchmark measurements are normalized against the JavaScript version of each corresponding individual benchmark in JavaScript running the exact same input, rather than the time taken to run the *unit benchmark*.

3.1 Distribution of Client Performance Scores

Figure 2 shows a histogram of the time a device needs to complete the execution of our reference benchmark in JavaScript, giving an overview of the range of TouchDevelop clients performance. The client *unit time* piggybacks on every synchronization request to the cloud, allowing us to examine how fast our client platforms are. The histogram shows three distinct classes:

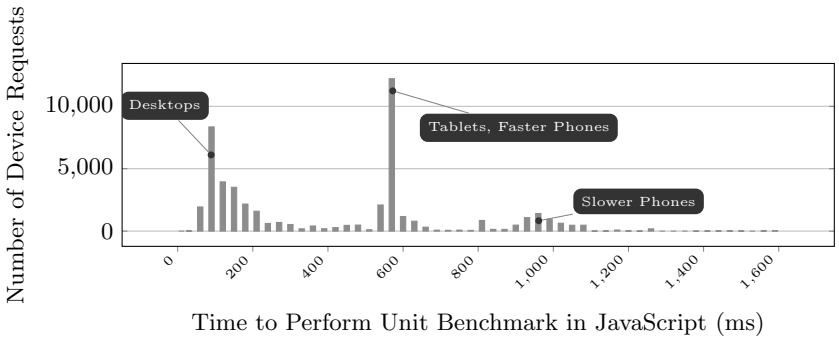

Fig. 2. Unit time histogram (50,000 client requests)

1. **Desktops**: With an average of 70ms to complete the JavaScript benchmark, these represent the fastest edge in the devices spectrum.
2. **Tablets and Faster Phones**: They have an average of 570ms to complete the benchmark and represent the latest generation smartphones and tablets.
3. **Low-end Phones**: They have a wider variation and greater diversity in models, but typically completes the benchmark in approximately 1 second, 10 times slower than desktops. Their worse performance is due to a combination of simpler hardware and JIT engines.

4 TouchDevelop compiler overview

The TouchDevelop compiler is the component that translates scripts written in the TouchDevelop language to pure JavaScript running on the following browsers: Internet Explorer 10+, Chrome 22+ for PCs, Macs and Linux, Firefox 16+ for PCs, Macs and Linux, Safari 6+ for Macs, Mobile Safari on iOS 6+ for iPad, iPhone and iPod Touch and Chrome 18+ for Android. Figure 3 shows a diagram with an overview of how scripts are executed.

The complete software stack involves two layers of translators, the first translating TouchDevelop scripts to JavaScript, and the second translating JavaScript to machine code. We use a black-box approach to the second layer and we do not focus on investigating its internals, but we wish to infer its capabilities by analyzing performance results. This section discusses only the first layer.

The script on the left-hand side of Figure 3 is the input script written by the user. Since the script code can call asynchronous functions (e.g. consume a web service), the *Execution Manager*, the component responsible for ensuring correct script execution, must remember the context of the call in order to resume script execution when the request response arrives. However, there is no support for direct jump to a specific point of the code in JavaScript. To overcome this issue, the Execution Manager splits the script code into several separate native JavaScript functions and executes them in a continuation-passing style [11].

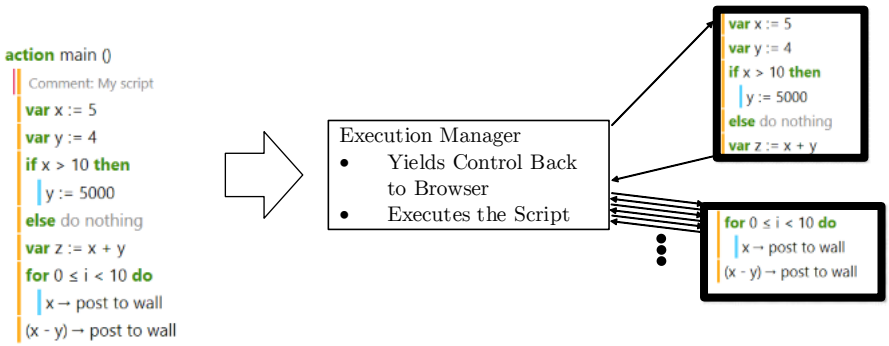

Fig. 3. Overview of TouchDevelop scripts execution

In the script, every point that is a target of a jump starts a new JavaScript function. Besides asynchronous calls, this is also true for loop structures because the Execution Manager must also ensure that the browser user interface (UI) update stack runs periodically, which does not happen if a loop structure runs for too much time without returning to the Execution Manager. In this case, the

UI may look frozen or, in an even worse scenario, the browser may terminate the script (after possibly asking the user), which is undesirable, particularly for games. The Execution Manager avoids this situation by deciding the next program segment to run; if a time budget is exceeded, it yields control back to the browser by means of a call to a `setTimeout` function to resume script execution later.

Owing to the lack of a jump construct in JavaScript, the continuation-passing style execution is a common language implementation technique and we targeted two optimizations at improving this kind of execution. The next subsections present all three code transformations we employed to optimize the execution of the scripts compiled to JavaScript code.

4.1 Safety Checks Elimination

Prior to every use of a value in the TouchDevelop language or in any other language where sanity checks must be performed, the value must be checked for `undefined` references (see Figure 4). In the case of TouchDevelop, where first-time programmers are the language target audience, the detection of uses of the `undefined` value makes it easier to understand and spot bugs. The removal of these safety checks can propagate the error inside a runtime function and cause crashes outside the scope of the TouchDevelop script, that is, errors in the JavaScript run time library that intimidates novice programmers unaware of the underlying infrastructure.

```
function ok1(a0) {
    if (a0 == undefined)
        TDev.Util.userError("using invalid value");
}
```

Fig. 4. Code excerpt for the safety check

Figure 4 shows a separate function to check for undefined references. We put the code in a separate function to help us distinguish this code in our profiler; inlining the calls to this function has no difference in performance.

Figure 5 shows the results of profiling, on an Internet Explorer 10 desktop platform, of the execution of the Mandelbrot program from the CLBG implemented as a TouchDevelop script. Mandelbrot spends most of its time in a loop body calculating values of the pixels of a fractal image. Function `arun6` is this loop body and, therefore, corresponds to time spent executing the actual algorithm.

All other functions are execution overhead. The `ExecutionManager` entry is the time spent inside the Execution Manager while it is giving back control to the browser or to the next script fragment scheduled. The `ok1` and `ok2` functions are safety checks for 1 and 2 arguments operations respectively. Therefore, 72%

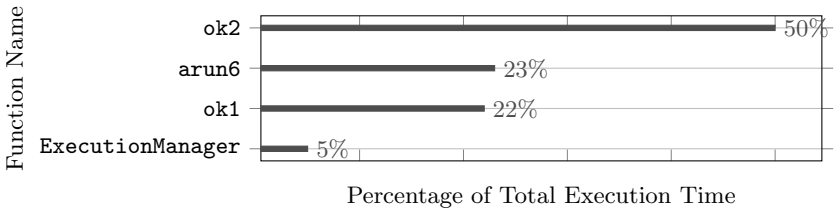

Fig. 5. Profiling of the Mandelbrot benchmark script in Internet Explorer 10 for a desktop machine.

of the script execution time is spent checking whether values are undefined for the Mandelbrot when running on Internet Explorer.

This motivated the construction of analyses passes to remove unnecessary, redundant checks for which we can either prove that the tested value is never undefined or that has already been checked in the past and was not changed since then.

4.2 Stack Frame Bypass

Recall that if the script calls an asynchronous function, the Execution Manager needs to remember the point where the script stopped in order to resume the execution when the response comes, and how this can be addressed by the continuation-passing style of execution. It also needs to remember all of the caller action local variables. This context-saving performed by the Execution Manager requires the maintenance of a data structure to hold the call stack with stored local variables for the current action.

To allow this, each time the script needs to call an *action*, the TouchDevelop analogue for a function, the Execution Manager first needs to build an object to hold all locals of this action and then call the first function fragment to start its execution. Furthermore, the explicit stack frame causes an additional overhead: each local read and write translates to JavaScript object accesses instead of a JavaScript local variable access.

However, if the action does not call other actions and does not have loop structures, there is no point in building expensive, explicit stack frames because there is no need to resume execution of the action: it executes once and exits back to the caller. It is possible to build a call graph and remove the stack frame from leaf functions with these properties. When this is done, the script can bypass the Execution Manager and call the leaf function directly, as it would call a JavaScript helper function, since the execution manager does not need to instantiate a special stack anymore.

Figure 6 shows the call graph construction where we can see that Action F is a simple leaf function that can be emitted as a native JavaScript function. An important observation is that if an action only calls other actions that don't need context and it does not have loop structures or calls to asynchronous functions,

it also does not need a context itself. To implement this, we employ a bottom-up analysis of the call graph, which enables us to remove the stack frame at multiple levels, not only leaf functions.

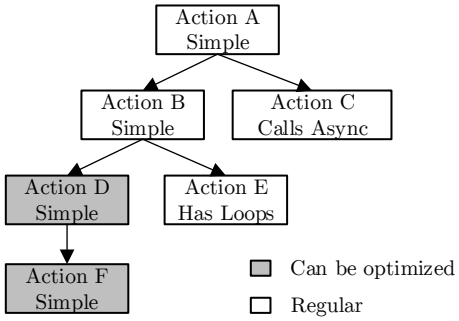

Fig. 6. A call graph identifying optimization opportunities for actions whose stack frame can be removed.

4.3 Block Chaining

The Execution Manager is an expensive mechanism in script execution because after a script fragment finishes, it hands over control back to the Execution Manager along with an indication of the next fragment to execute, which means the regular program flow always involves visiting the Execution Manager several times. This is especially true for loop constructs because they involve going back to some previous point in the script and this is accomplished by isolating the loop body into a separate JavaScript function that will be called every iteration.

At the end of each loop iteration, it must return back to the Execution Manager that in turn calls the fragment again to execute the next iteration. It is not possible to bypass the Execution Manager by emitting a native `for` or `while` construct in JavaScript because if the loop body makes an asynchronous call, it is no longer possible to resume execution to the next program point. Furthermore, giving control to the script for too much time, for instance, over many iterations of a loop, can delay the browser UI update thread and make the app looks unresponsive.

For a loop-intensive benchmark like the Pfannkuchen program, which repeatedly calculates permutations using a complex loop structure, this mechanism generates a considerable overhead. Figure 7 shows the profiling of this program running on Chrome 29 for desktops, and we see that the Execution Manager actually spends more time than the application itself.

Figure 8 presents a technique to avoid excessive returns to the Execution Manager by chaining fragments execution: instead of returning the next fragment to execute, a fragment can call the next fragment itself, bypassing the Execution

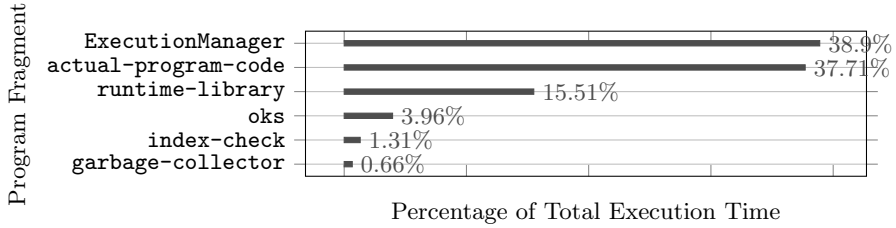

Fig. 7. Profiling of the Pfannkuchen benchmark script in Chrome 29 for a desktop machine.

Manager. To avoid that a really long loop takes control of the thread making the app unresponsive, we add a *trip count* to mark how many iterations skipped the Execution Manager and once a threshold is met, it finally returns to the Execution Manager. Notice that this parameter affects the call nesting level and should be tuned by platform, since some systems, most notoriously the Mobile Safari browser, implement a very shallow call stack.

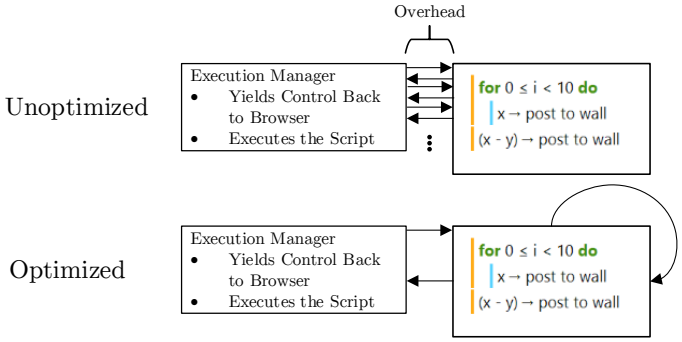

Fig. 8. Chaining blocks of execution to bypass the Execution Manager

Even taking care of the trip count, this technique can reduce responsiveness of the application unnecessarily. It is important to have the crowdsourced performance measurements to know where it is profitable to apply this optimization. The next section dives into the crowdsourced performance results and discuss the effectiveness of all these optimizations.

5 Results

We show the crowdsourced compiler flag recommendation system in practice with an experiment to test the performance of the three optimizations implemented for JavaScript code emission. Users of TouchDevelop were prompted to

help with benchmark measurements and, once accepted, a single benchmark with random flags ran on their platform and the results were uploaded back to the cloud. We collected more than 1,000 measurements, allowing us to draw clear conclusions for 8 different platforms. The results appear in Figure 9.

Platform	Safety Checks Elimination	Stack Frame Bypass	Block Chaining	Suggested Flags
Tablet with Windows 8 and IE 10	5.8	5.8	5.6	**sfb**
Desktop with Windows 8 and Chrome	1.1	1.2	1.1	**-**
Desktop with Windows 8 and IE 10	2.6	2.6	2.6	**sfb**
Desktop with Windows 7 and Chrome	1.1	1.4	1.1	**f**
Tablet with Windows 8 and IE 11	4.5	5.0	6.0	**sfb**
Cellphone with IE 10	1.5	1.5	1.7	**sfb**
Desktop with Windows 7 and Firefox	0.9	0.8	1.6	**b**
Desktop with Windows 7 and IE 10	1.3	1.3	1.3	**-**

Fig. 9. Color-coded recommendation table that suggests which flags to apply on each client browser platform.

The second line shows that the Windows 8 with Chrome platform has, on average, 10% performance improvements after *Safety Checks Elimination*, 20% after *Stack Frame Bypass*, 10% with *Block Chaining* and therefore no special flags are recommended for this platform. In order to show that an optimization is really important, the crowdsourced data must show that the average improvements for a given platform surpass 30%. We see this scenario for a Tablet with Windows 8 and IE 11: programs run 4.5 times faster after removing safety checks, 5 times faster after bypassing the stack frame whenever possible and 6 times faster with block chaining. However, we see a 20% performance decrease for Windows 7 with Firefox, showing that changing the code can actually be worse for some platforms and, therefore, the importance of crowdsourcing the performance of optimizations, checking whether we have real improvements.

In order to understand why the crowdsourced data lead to these conclusions, the next subsections describes in detail experiments on a single desktop platform, showing what happens with each browser after each of our optimizations are turned on. Finally, we show how the improvements on a Microsoft Surface RT Tablet platform look like.

5.1 No optimizations

Figure 10 presents the performance figures for our benchmark implemented in TouchDevelop compared against optimized hand-crafted JavaScript code, for different JIT engines. For example, the program *Binary Trees* in TouchDevelop runs 46.3 times slower than the same algorithm implemented in JavaScript on Chrome 27. Slowdowns of this magnitude are expected because of the runtime mechanism for TouchDevelop scripts, which is continuously interrupting script

execution to yield control back to the browser when it is necessary. However, higher slowdowns are a consequence of a performance bottleneck.

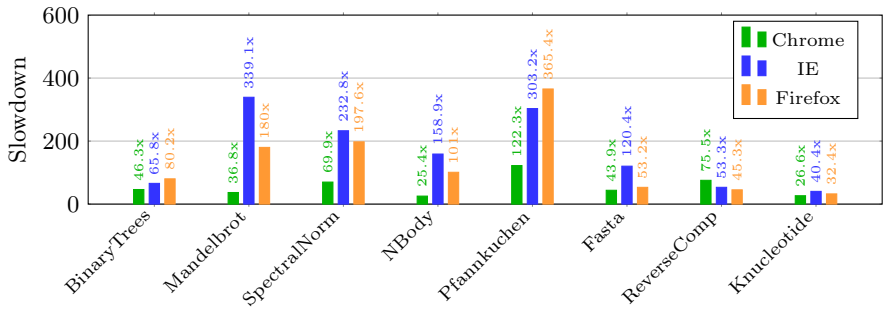

Fig. 10. The slowdown of running each benchmark as a TouchDevelop script, when compared to optimized JavaScript code.

The JavaScript optimized performance is only used as a reference and as an upper bound for performance. We focus on the difference of performance between the different JIT engines. Perhaps the most notorious performance result is that of Mandelbrot, a small program that fits almost completely in Figure 11. Its purpose is to calculate a fractal image, and for each pixel of the image it uses a formula to determine whether the pixel is black or white. Its performance running on a desktop with Chrome 27 is 36.8 times slower than the JavaScript version running in the same environment, while on Internet Explorer 10 the slowdown is 339.1 times and on Firefox 21 it is 180 times.

The cause of such large performance differences amongst different JIT engines is a consequence of the different compilation schemes employed by each one, and we need to be aware of these idiosyncrasies and handle them when optimizing for performance. The fact that Internet Explorer 10 runs this script with a slowdown of 339.1 means that either the JavaScript baseline version is too fast or the TouchDevelop version is too slow, when compared to other browsers. In both cases, it is clear that out compiler fails to extract the performance that this browser can deliver for this code fragment as good as we do it for Chrome. Nevertheless, the JavaScript execution time differences of Mandelbrot for both Chrome and Internet Explorer are negligible, showing that the problem is really a bad interaction of our generated JavaScript code and Internet Explorer 10.

The programs *Spectral Norm*, for calculating eigen values, *N Body*, for performing physics simulation and *Pfannkuchen*, for calculating the maximum number of permutations in a math riddle, all suffer similar performance differences between JIT engines.

```
for 0 ≤ i < h do
  var Ci := i * y fac - 1
  for 0 ≤ j < w do
    var Zr := 0
    var Zi := 0
    var Tr := 0
    var Ti := 0
    var Cr := j * x fac - 1.5
    var run loop := true
    for 0 ≤ k < 50 do
      if run loop then
        Zi := 2 * Zr * Zi + Ci
        Zr := Tr - Ti + Cr
        Tr := Zr * Zr
        Ti := Zi * Zi
        if Tr + Ti > 4 then
          run loop := false
        else do nothing
      else do nothing
    if run loop then
      pic→ set pixel(i, j, colors→ white)
    else
      pic→ set pixel(i, j, colors→ black)
```

Fig. 11. Main loop of the Mandelbrot algorithm implemented in TouchDevelop, accessible via *https://www.touchdevelop.com/iyyydbkw*.

5.2 Safety Checks Elimination

Figure 12 presents the results of the safety checks elimination, an optimization we wrote for the TouchDevelop compiler, and its effects on different JIT engines.

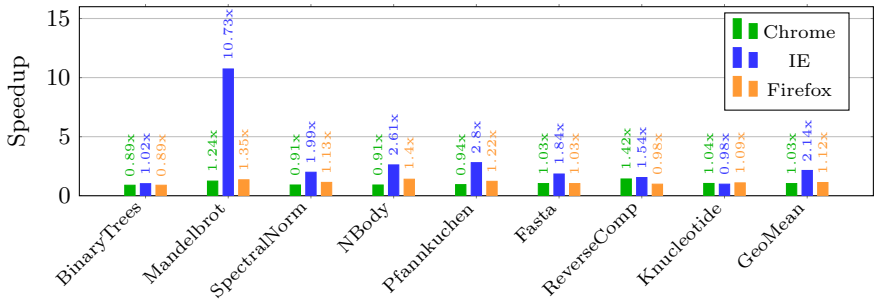

Fig. 12. Elimination of safety checks: Speedups over baseline with no optimizations.

The graph now shows the speedup of the optimized script code versus the unoptimized version. We see that one of the pathological cases, Mandelbrot, got its performance substantially improved (10.73 times faster) in Internet Explorer 10, bringing its slowdown, when compared to the native JavaScript version, to 31.6 times for Internet Explorer. Chrome 27 executes the same program with 29.7 times of slowdown, while without optimizations it executed with 36.8 times of slowdown.

The elimination of safety checks has almost no effect on Chrome, but it is really important for Internet Explorer, showing that the knowledge of the underlying platform that is running our script is crucial to drive which optimizations our compiler should apply.

5.3 Stack Frame Bypass

Figure 13 presents the effect of bypassing the creation of a separate stack frame for actions in which it is not necessary to have one. The graph shows the cumulative effect of applying both the elimination of safety checks and stack frame bypass. The greatest speedup remains that of Mandelbrot thanks to the elimination of safety checks. The stack frame bypass affects only Spectral Norm, which is a program whose inner loop depends on calling a helper action and therefore exercises this kind of bottleneck. However, the platforms see uneven improvements. Firefox 21 benefits the most out of this optimization, with an speedup of 9.11 times, while Internet Explorer 10 had 2.83 times, the lowest improvement, and Chrome 27 had 3.92 times.

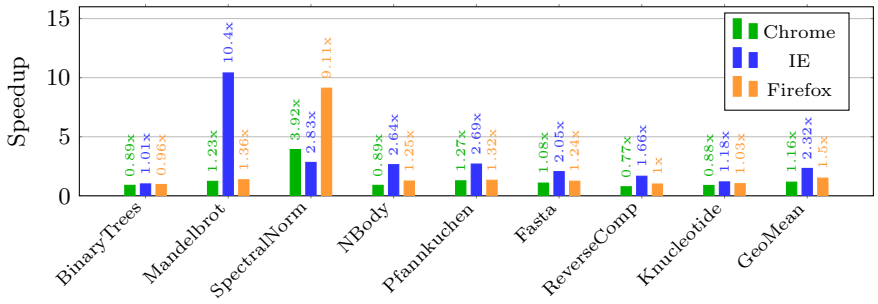

Fig. 13. Adding stack frame bypass: Speedups of safety checks elimination added with stack frame bypass over baseline with no optimizations.

For Firefox 21, Spectral Norm started with a slowdown of 197.6 times and, after this optimization, finished with a slowdown of 21.7 times, one of the lowest slowdowns for TouchDevelop scripts execution.

5.4 Block Chaining

Figure 14 shows the final improvements of all optimizations, including *block chaining*, when compared with the baseline without optimizations for a desktop computer. The *block chaining* boosts Mandelbrot speedup in Internet Explorer 10 to be 21.19 times faster, making its original slowdown, when compared to pure JavaScript, to be of 16 times, as opposed to 339.1 times without optimizations.

The speedup of Mandelbrot for Firefox 21 boosts from 1.36 times to 8.78 times faster, showing that, for Firefox, the block chaining mechanism is much

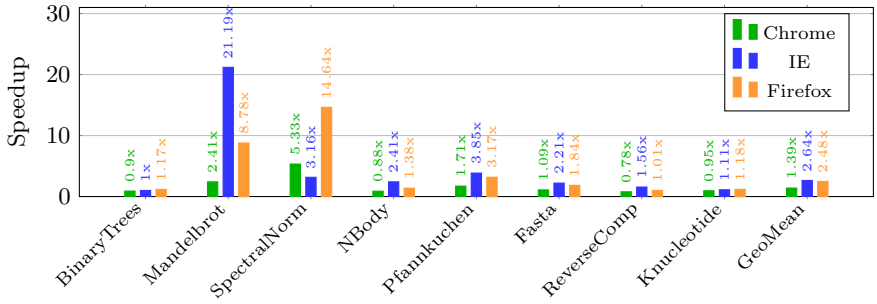

Fig. 14. Adding block chaining: Speedups of all optimizations over baseline with no optimizations.

more important than the elimination of safety checks. The block chaining in Firefox is also responsible for making Spectral Norm 14.64 times faster, doubling the gain obtained by *stack frame bypass*.

5.5 Surface RT with IE 11

So far, we have presented the effects of three code optimizations when generating JavaScript code for desktop browsers. However, the main audience for the TouchDevelop project are mobile users, since it is a touch-friendly integrated development environment. The combination of the software JIT methods with the underlying simpler hardware platform creates yet another effects on the results. The final effect of applying all three optimizations described in this paper when running on a Microsoft Surface RT tablet with Internet Explorer 11 appears in Figure 15.

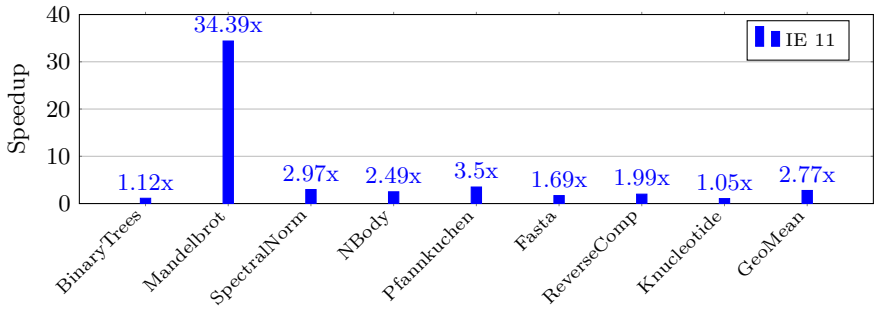

Fig. 15. All optimizations speedups over baseline performance for Surface RT with Internet Explorer 11.

In general, we observe that these optimizations are more important for the Surface RT platform: the geometric mean of the measured speedups among all programs is 2.77x, while for a desktop machine it is 2.64x. The largest improvement is still Mandelbrot, but it is now 34.49 times faster, while on a desktop the maximum speedup seen in Mandelbrot is 21.19x. The simpler low-power ARM-based hardware platform is more sensitive to code improvements in comparison with a power-hungry Intel out-of-order core, which can extract instruction-level parallelism and can compensate for a lower quality code emission by the JIT engine.

6 Related work

In this work we study how we can infer optimization flags based on crowdsourced performance data for a compiler that produces JavaScript code. Since JavaScript performance is largely dependent on the JIT compilation techniques employed by the JavaScript interpreter, it is important to know which browser will run this code. However, we assume no knowledge about the underlying JIT compilation mechanism – instead, we expect to draw all necessary conclusions from our collected data. On the other hand, there are several other studies that focus on tuning the underlying JavaScript JIT compilation mechanism to improve performance [12,21,22,23,24] or on selecting the best JavaScript framework to program with [17]. For instance, Lee and Moon [22] study, for mobile web browsers, how the JIT engine can be turned on or off in order to avoid waiting for the compilation of a code that may be not frequently executed enough to pay off the compilation time. Furthermore, mobile web browsers are such an important cellphone use case that they deserve a specific study on how to render pages with the minimum use of battery: Zhu and Vijay [31] analyze how to leverage heterogeneous multiprocessor systems to render mobile web pages fast enough to the user while maximizing power efficiency. Notice that although our data collection mechanism cannot receive power information from our users, for homogeneous systems, our recommendation system also improves power efficiency when it reduces total script execution time.

Another important related field of study is in determining the best set of optimizations to apply on a given compilation unit for traditional, static compilers that does not rely on cloud support [13,14,16,19,20,25,26]. These works differ from ours because they are targeted at traditional compilers rather than a web-based compiler, and they focus on producing efficient assembly code rather than JavaScript that runs on top of a JavaScript interpreter. Since, in this scope, this problem involves deciding between tens of optimizations and the ordering between then, it is a difficult problem even for a single well-known platform. While compiler users traditionally have been using a fixed set of optimizations to all programs, Cavazos et al. [13] were able to reduce the Jikes virtual machine execution time on the SPECjvm98 benchmark by 29% on average by employing machine learning techniques to train the Jikes system to recognize methods and decide which subset of optimizations to apply and its order. Pan et al. [25],

Haneda et al. [19] and Pekhimenko et al. [26] also investigate methods to automatically find a good subset of optimizations to apply to a given program.

Perkhimenko et al. apply a similar technique of Cavazos to a commercial static compiler, reporting a compilation run time speed up by a factor of at least 2. To do this, a feature vector – characteristics that describe a method – is computed from a program at compile time (statically) for the commercial compiler Toronto Portable Optimizer (TPO). They extract instruction types and loop-based parameters to describe methods. Our approach, on the other hand, does not suggest flags per program, but it does per platform because we are primarily concerned with dealing with a large number of client platforms, which must be addressed before each program is fine-grainedly tuned. In our technique, we determine the overall optimization efficiency based on the performance reports of a benchmark set. If the benchmarks show improvements in total run time for a given platform, the selected flags are suggested to be used for all programs of this platform. Notice that the benchmark was selected to exercise the language performance bottlenecks.

Perhaps the work with a greater similarity to ours is the GCC MILEPOST project [16], an adaptive compiler framework that was created for research purposes. To our knowledge, the MILEPOST project is the first attempt to use tuning technology using a crowdsourced database to create real-world self-tuning machine learning enabled static compilers. However, their work is targeted at lower level compilation rather than JavaScript code emission, and their challenges are quite different since they do not have to run code on top of an existing JIT infrastructure. Our work, on the other hand, is targeted at learning how to generate efficient code for different browsers and JIT engines using crowdsourced data for a modern, device independent browser-based programming language and development environment. MILEPOST relies on machine learning techniques to train the traditional open-source compiler GCC [15] in how to best optimize programs. Fursin et al. [16] were able to learn a model that improved the performance of the MiBench [18] benchmark by 11%.

7 Conclusion

We present a system that collects web clients performance data to build a crowdsourced compiler flag suggestion system in the cloud, helping the compiler perform the appropriate optimizations for a given platform. We also show a set of optimizations that address common language implementation issues when targeting JavaScript. Extracting performance and generating quality code for JavaScript is quite challenging, since each JavaScript engine behaves in a different way.

We implement and test these concepts in a real web-based programming environment used by tens of thousands of users, TouchDevelop, allowing us to better explore the crowdsourcing and cloud dimensions of this project. We enhance the TouchDevelop compiler, which translates TouchDevelop scripts to JavaScript, with three new optimizations.

We then present data from more than a thousand users that collaborated with the project, showing a scenario where optimizations, on average, extract 5x speedups in a tablet with Windows 8 and Internet Explorer 11 but reduces performance by 20% in a desktop with Windows 7 and Firefox. This is a consequence of running code on top of complex Just-in-Time compilation engines, which already apply its own optimizations and can make undisclosed code transformations. The crowdsourcing approach allows us to detect such scenarios and disable unfruitful optimizations in a per-platform basis by simply asking the cloud the best set of flags for a given client.

In this work we assume no knowledge about the underlying JavaScript engine. We rely on the crowdsourced performance data in order to overcome the difficulties of increasing the performance of JavaScript and come up with an adaptive compiler that applies a custom set of optimizations for each web client. This automatic compiler flag suggestion system is able to cope with a wide variety of more than 30 different client platforms without any manual effort.

As future work, we intend to leverage an existing system of crowdsourced profiling of the scripts to also record the effects of a particular optimization on the average performance of real-world programs in the field.

References

1. Develop High Performance Windows 8 Application with HTML 5 and JavaScript. http://blogs.msdn.com/b/dorischen/archive/2013/04/26/develop-high-performance-windows-8-application-with-html5-and-javascript-best-practices-amp-tips.aspx.
2. Google Web Toolkit Page. http://www.gwtproject.org/.
3. Mozilla SpiderMonkey JavaScript Engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
4. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/.
5. The Dart Language Web Page. https://www.dartlang.org/.
6. 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.
7. The TypeScript Language Web Page. http://www.typescriptlang.org/.
8. The WebKit Open Source Project. http://webkit.org/.
9. V8 JavaScript Engine. http://code.google.com/p/v8.
10. V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: a low-level virtual instruction set architecture. In *MICRO36*, 2003.
11. A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992.
12. M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA 2010*. ACM.
13. J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA 2006*. ACM.

14. K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *LCTES '05*, 2005.
15. Free Software Foundation, Inc. *Using the GNU compiler collection*, Mar 2013. For GCC version 4.9.0.
16. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, et al. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, 2008.
17. A. Gizas, S. P. Christodoulou, and T. S. Papatheodorou. Comparative evaluation of javascript frameworks. In *21st international conference companion on World Wide Web*, 2012.
18. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IISWC 2001*. IEEE.
19. M. Haneda, P. M. Knijnenburg, and H. A. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT 2005*. IEEE.
20. K. Hoste, A. Georges, and L. Eeckhout. Automated just-in-time compiler tuning. In *CGO 2010*. ACM.
21. S. Jeon and J. Choi. Reuse of JIT compiled code in JavaScript engine. In *27th Annual ACM Symposium on Applied Computing*, 2012.
22. S.-W. Lee and S.-M. Moon. Selective just-in-time compilation for client-side mobile javascript engine. In *CASES 2011*. ACM.
23. S.-W. Lee, S.-M. Moon, W.-J. Kim, S. jin Oh, and H.-S. Oh. Code size and performance optimization for mobile JavaScript just-in-time compiler. In *2010 Workshop on Interaction between Compilers and Computer Architecture*.
24. J. K. Martinsen, H. Grahn, and A. Isberg. Using speculation to enhance javascript performance in web applications. *IEEE Internet Computing*, 17(2):10–19, 3 2013.
25. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO 2006*. ACM.
26. G. Pekhimenko and A. D. Brown. Efficient program compilation through machine learning techniques. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, page 335, 2010.
27. G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *OOPSLA 2011*. ACM.
28. S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.
29. N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *ONWARD 2011*. ACM.
30. A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *SPLASH 2011*. ACM.
31. Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA 2013*. IEEE.