Overhead of Deoptimization Checks in the V8 JavaScript Engine

Gabriel Southern and Jose Renau

Dept. of Computer Engineering, University of California, Santa Cruz
Email: {gsouther,renau}@ucsc.edu

***Abstract*—Just-in-time compilation for dynamic languages uses type speculation to generate optimized code for frequently executed code sections. However, even the high-performance code requires checks to ensure that assumptions made during code generation are not violated. If the assumptions are violated then the program reverts to executing unoptimized code. This process is called deoptimization, and the checks are called deoptimization checks.**

We analyzed the frequency of conditional deoptimization checks and the performance impact of the associated conditional branch using the V8 Javascript engine running the Octane benchmarks. Over half of the benchmarks have at least 5 checks per 100 instructions, and 1 benchmark has as many as 19 checks per 100 instructions.

The performance impact of the checks varies significantly depending on the microarchitecture of the CPU that is running the code. On average the conditional branch used by the checks accounts for over 6.2% of the benchmark's dynamic instructions. Removing these checks provides only a 2.2% performance improvement when running on a high performance Intel CPU, but a 4.6% performance improvement on an Intel CPU optimized for low power consumption.

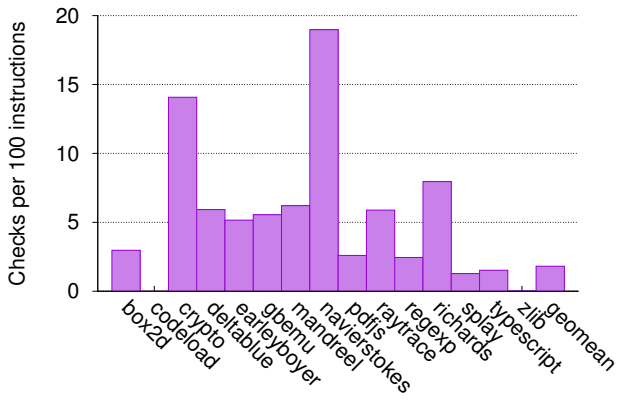

Fig. 1. Frequency of deoptimization checks for the Octane benchmarks when executed with the V8 JavaScript engine. Eight of the 15 benchmarks have more than 5 checks per 100 instructions, and many of the checks are comprised of multiple instructions.

I. Introduction

Dynamic languages such as JavaScript, Python, and Ruby have become extremely popular in recent years. Applications written in dynamic languages have historically executed much more slowly than those written in statically typed languages, in part because variable type information is only known at runtime, and not at compile time. However, in recent years the performance of JavaScript runtime and compilers has improved dramatically and JavaScript is widely used in client and server applications that require high performance.

JavaScript implementations attain their high performance by using multi-tiered just-in-time (JIT) compilation combined with type speculation [6], [12], [16], [17]. The initial tier quickly parses JavaScript source code and executes it with an interpreter or a fast compiler that does not perform any optimizations. As this code runs it stores the types of variables used by a function in inline-caches using techniques first developed for Smalltalk [7] and Self [9]. After type information has been collected, frequently executed code is recompiled with an optimizing compiler, which assumes that variable types will not change even though this is not guaranteed by the language semantics. In order to guarantee that these assumptions are not violated, the compiler inserts checks to ensure that the assumptions about types still hold. If the condition expected by the checks fails to hold then the function stops running the optimized code and switches back to the unoptimized version. The process is called deoptimization[1], and the checks are called deoptimization (or deopt) checks.

The cost of each conditional deoptimization check is relatively small, typically ranging from 1 to 3 instructions. An example of a common check is the comparison of an object's pointer to its type with a pointer to the expected type. The comparison is followed by a conditional branch that branches to deoptimization code in case the pointers do not match. Although the cost of each check is small, the checks can occur very frequently for applications that run mostly in optimized mode. Figure 1 shows the frequency of deoptimization checks for the Chrome V8 JavaScript engine when running the Octane benchmark suite [1]. Over half of the benchmarks execute at least 5 checks per 100 instructions, and many of the checks are comprised of more than 1 instruction.

In this paper we analyze the type and frequency of deoptimization checks used in the Chrome V8 JavaScript engine, as well as the performance overhead associated with these checks. Although our analysis focuses on Google's Chrome V8 JavaScript engine, all major JavaScript engines including those from Apple, Mozilla, and Microsoft have deoptimization checks, and deoptimization checks are present in JITs for Python, Ruby, and other dynamic languages. We believe

[1]Also sometimes called "bailout"

that the insights gained from studying deoptimization check overhead for a specific JIT implementation can be useful in other contexts since speculative optimization is a widely used technique in JITs for dynamic languages.

We identified four main types of deoptimization checks that are used in Chrome V8 and characterized the frequency of checks when running the Octane Benchmark suite. We also analyzed the performance overhead of the conditional branches used for the checks by running the benchmarks on four different real systems with checks enabled and disabled. We found that modern wide issue out-of-order processors are usually able to absorb the overhead of executing the extra check instructions with minimal performance overhead. However, narrower issue systems suffer a larger slowdown that corresponds with the instruction count overhead.

This paper's contributions are:

- Characterizing the performance overhead of deoptimization checks for the Chrome V8 JavaScript engine on four different machines,
- Showing that performance overhead does not always correlate with instruction count overhead, but rather depends on a combination of the workload and the system micorarchitecture; and
- Categorizing the types of deoptimization checks and characterize their execution frequency when running optimized code.

The rest of this paper is organized as follows: Section II provides background information about JITs for dynamic languages, the V8 JavaScript engine, and the Octane benchmark suite that we used for our experiments; Section III shows the types of frequency of deoptimization checks that we observed; Section IV provides an analysis of performance overhead associated with deoptimization checks; Section V surveys related work; and Section VI concludes.

II. Background

A. Dynamic Languages

Dynamically typed languages do not require developers to declare types of variables in the program source code; instead, types are determined at runtime based on the state of the running program. These languages are often used to improve programmer productivity [11], [15], even though they can have a significant performance overhead.

The simplest way of implementing a runtime system for a dynamic language is to use an interpreter that directly executes statements in the program source code. Each time an operation is performed, the interpreter checks the type of the variables used in the operation and selects the method that is appropriate based on the variable types.

To make this more concrete, consider the snippet of JavaScript code shown in Listing 1. In this example the function `twice` has different behavior depending on the type of its input parameter. When given the numeric argument 1 it adds 1 and 1 and returns the value 2. But when given the string argument "1" it concatenates "1" and "1" and returns the string "11". When an interpreter executes the function `twice` it first determines the type of the input parameter `arg` and then it chooses the correct operation to perform based on this type.

Listing 1. JavaScript dynamic dispatch example

```
function twice(arg) {
  return arg + arg;
}
res = twice(1);     // res is number 2
res = twice("1");   // res is string '11'
```

In general interpreted languages suffer a large performance overhead compared to statically typed compiled languages. Part of the overhead is associated with the dispatch loop, which must continually check the type of each variable and then execute the correct functionality based on the variable type. The other part of the overhead is caused by the fact that the type checks prevent many common compiler optimizations.

Despite the fact that these languages allow variables to change their type at runtime, often the runtime behavior is very predictable. Smalltalk [7] and SELF [9] pioneered techniques using inline caches and dynamic compilation to improve performance, and these techniques are used in the most popular JavaScript systems today [6], [12], [16], [17].

To understand how this works, consider the example in Listing 2 and imagine that the array `input` contains many integers. During the first iteration of the `for` loop the runtime system looks up the appropriate operation (addition) for the `twice` function based on the fact that its input parameter is a numeric type. At the same time the runtime system saves this information in an in-line cache to simplify the method lookup process in case the `twice` function is called with an integer argument again.

During future iterations of the `for` loop the runtime system will be able to use information cached in the in-line cache when determining how to execute the `twice` function. However, even with this optimization there is a significant performance overhead compared to the code that a statically compiled language like C++ could generate given the knowledge that the parameter for the function is always an integer.

To further improve performance the runtime system can generate an optimized version of the `twice` function that assumes that the `arg` parameter is always an integer. This optimized code does not need to check the inline cache to determine what method to execute; instead, the compiler can perform integer addition using a single machine code instruction.

Listing 2. Inline cache example

```
function twice(arg) {
  return arg + arg;
}
var input  = [1,2,3,4, . . .];
var output = [];
for(var i of input) {
  output.push(twice(i));
}
```

However, this only works if the `twice` function is called with an integer argument. And the JavaScript semantics do not guarantee this—instead they allow any type of argument. The runtime systems solves this problem by inserting a

deoptimization check before using the `arg` parameter. The deoptimization check is called at runtime and ensures that the `arg` parameter has the correct type for the optimized code. If `arg` parameter is not an integer then the check fails and triggers a deoptimization event. The deoptimization restores the state of the program to one where the unoptimized compiler can resume code execution and look up the correct method based on `arg`'s type.

In this paper we focus on understanding the frequency and performance overhead of the deoptimization checks that are inserted in the optimized code by the compiler. Speculative compilation with deoptimization checks is used by many runtime systems for dynamic languages. We chose to focus on the V8 runtime system that runs JavaScript code in order to study a specific language and runtime system combination. In the next section we provide some background about V8 and why we chose it as the runtime system to use for our study.

B. V8 JavaScript Engine

The V8 JavaScript engine was created as part of Google's project to develop the Chrome web browser and was first released in 2008. However, V8 itself was developed as a standalone open source project. It remains under active development today and is still used in the Chrome browser. It is also used by node.js, which is widely used to run server side JavaScript code. The initial version of V8 only had a single compiler, but in 2010 a new version with an optimizing compiler was released. The optimizing compiler made use of speculative type information stored in in-line caches and required deoptimization checks. The initial compiler is called "full-codegen" and the optimizing compiler is called "crankshaft." In 2014 Google added a third compiler named TurboFan to the V8 engine. This compiler attempts to improve performance by applying the "sea of nodes" [5] techniques. However, currently all the Octane benchmarks except for Zlib spend the majority of their time executing code generated with the Crankshaft optimizing compiler.

Our study was motivated by a desire to understand the performance overhead the frequently executed but rarely taken deoptimization checks impose on dynamic languages and to consider possible hardware extensions to reduce this overhead. As a result we wanted to study a runtime system that delivers high performance, since a low performance system may imply the need for hardware extensions, when in fact better software engineering could remove overhead without changing the CPU architecture. JavaScript is one of the most widely used dynamic languages, and developers have invested more effort in developing high performance runtime systems for JavaScript than any other dynamic language in use today. V8 and Firefox are available as open source projects that run on Linux, and compete for the best performance results on widely used JavaScript benchmarks. Either would have been fine to use for our analysis. We chose V8 in part because of its availability as a standalone project separate from the Chrome browser it is used with.

C. Octane Benchmarks

Another choice we had to consider was what benchmarks to use for our study. There are many benchmarks suites available and in our initial characterization work we gathered results from the JetStream, Massive, Kraken, and SunSpider suites, in addition to the Octane suite. SunSpider is very old and has effectively been replaced by JetStream. Our tests with SunSpider showed relatively few deoptimization checks, but that is largely because the benchmarks are so short that they spend little time executing optimized code. Our results from Kraken, Massive, and Jetstream were in roughly the same range as results we saw from Octane.

The Octane benchmark suite was developed by Google largely to help facilitate the development of the V8 JavaScript engine. As a result it is somewhat of an optimization target. However, our goal was to evaluate overhead of deoptimization checks in cases where the software had been tuned to get the best possible performance. So we chose it as the benchmark suite to use when collecting detailed performance results across a range of different systems.

III. Checks Categorization

In V8, deoptimization occurs whenever the runtime system transitions from executing optimized code to unoptimized code. There are two primary types of deoptimization events: conditional deopts and unconditional deopts.

When an unconditional deoptimization point is inserted in the optimized code, the program will always be deoptimized if it reaches this point. For example, if V8 were to generate optimized code for a hot loop, it might insert a deoptimization point at the loop exit. This way the optimizing compiler could compile just the body of the hot loop, and when the loop reached its exit condition the runtime would switch to unoptimized code where it could continue code profiling the code to identify new hot regions. This also means that the unconditional deoptimization check does not incur any additional overhead when executing *optimized code*. The check for the loop exit condition is needed regardless of whether the code that is executed outside of the loop is more optimized code, or if it is a deoptimization point. Deoptimization is a necessarily expensive operation because it causes the runtime to execute unoptimized code, which is much slower than optimized code.

In contrast, a conditional deoptimization point is inserted in optimized code when the compiler needs to ensure that some condition is true before continuing to execute the optimized code. For example, the compiler may have generated optimized code for a function assuming that the values the function is operating on are integers. However, the language semantics do not guarantee this, so the compiler must insert deoptimization checks, and if the function receives a value that is not an integer then it is deoptimized and V8 switches to using the baseline compiler. The conditional checks need to be very low overhead because they can occur frequently in the optimized code.

The V8 source code uses an enumeration type to define possible reasons for deoptimization. There are 64 possible

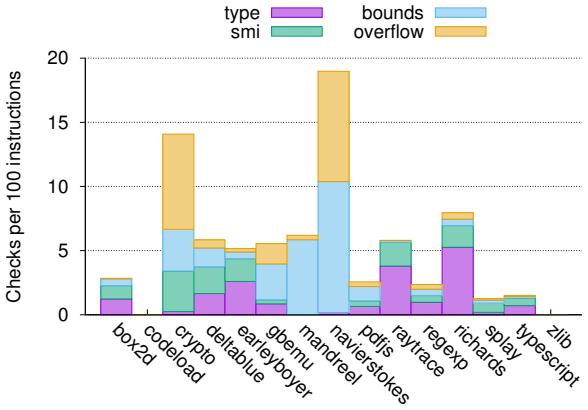

Fig. 2. Frequency of four different categories of deoptimization checks for each benchmark. Zlib and Codeload have less than 1 check per thousand instructions in any category and so their results appear as 0 on the plot.

reasons defined; however, these include reasons for both conditional and unconditional deoptimizations. We identified 24 of these deoptimization reasons that are used in conditional deoptimization when code is generated for the x64 target. We instrumented the code to count how frequently each of these deoptimization checks occur when running the Octane benchmarks. Of the 24 deoptimizations reasons that we instrumented, 7 of them occurred relatively frequently in at least some benchmarks while the remaining ones were executed a negligible number of times (less than once per thousand instructions). We grouped the frequently occurring checks into the following categories: type checks, small integer checks, bounds checks, overflow checks. Figure 2 shows the frequency of each of these types of checks, and we provide more details about each type in the following sections.

A. Type Checks

JavaScript uses prototypes rather than classes for creating objects, and the language itself does not define types. However, V8 creates "hidden classes" to group together objects created using the same prototype chain. This allows V8 to effectively treat objects as having a type, which allows for optimizations in accessing an object's fields.

Type checks are necessary to ensure that a variable has the type that the optimizing compiler expects. We categorized the following deoptimization reasons together as type checks: `kAccessCheck`, `kProxy`, `kHole`, `kWrongInstanceType`, `kWrongMap`.

The most frequently executed of these checks is kWrongMap and it is perhaps the easiest to explain. In V8, each object has a pointer to its "hidden class," which effectively defines its type. The "maps check" checks what hidden class the object is pointing to and makes sure that it matches the hidden class type that the compiler expects. If it matches, then the check succeeds; if not, then it fails and the code is deoptimized due to "wrong map."

The remaining deoptimization reasons that we have categorized as type checks are associated with different aspects of JavaScript semantics and V8 optimizations, but they all follow a similar pattern where a pointer value is compared to an expected value.

B. Small Integer (Smi) Checks

V8 uses tagged pointers as an optimization for representing "small integers" and storing them on the heap. This technique takes advantage of the fact that pointers to heap objects are aligned to 4-byte boundaries when they are allocated. As a result the lowest bit in the pointer to an object is always 0. However, when storing pointers to objects V8 sets this bit to 1 (since the correct value is always 0 no information is lost). Signed integers that can be represented in 31 bits or fewer are stored directly in the pointer field, and the lowest bit is set to 0.

The deoptimization reasons associated with Smi checks are `kSmi` and `kNotASmi`. In both cases the checks compare the lowest bit of the object's pointer to determine if it is a 0 or not.

C. Bounds Checks

Bounds checks ensure that an access to an array is within the array bounds. They work by comparing the array length to the index of the array access. If the index if larger than bounds limit then the check triggers a deoptimization with the reasons `kOutOfBounds`.

D. Overflow Checks

The final category of checks is overflow checks and these are used in a variety of arithmetic operations. These checks differ from the previous ones because no additional comparison operation is generated for the x64 architecture. Instead the checks use the result of the overflow condition code that is set implicitly. In cases where deoptimization is triggered, the reason is `kOverflow`.

IV. Performance Evaluation

In the previous section we characterized the types and frequency of conditional deoptimization checks. In this section we analyze the performance overhead associated with these conditional deoptimization checks by comparing the baseline performance metrics with metrics when the conditional deoptimization checks are skipped (i.e. not executed). In Section IV-A we explain how we were able to remove the checks and still have correct benchmark execution. Section IV-B describes our experimental setup for measuring performance metrics. Section IV-C compares the relative instruction count and execution time between the baseline system and the system that skips deoptimization checks. Section IV-D provides additional analysis looking at the IPC of these two configurations, and Section IV-E examines the branch prediction accuracy.

A. Experiment Methodology

Although deoptimization checks appear relatively frequently, they are almost never taken in the Octane benchmarks. These checks evaluate a variety of conditions, but all of them include a conditional branch. In order to estimate the overhead

of the conditional deoptimization checks we modified the V8 source code to remove the conditional branch when we could guarantee that it would never be taken. This allowed us to compare the performance of the baseline V8, which has deoptimization checks, with the performance when the checks are skipped. We selected which checks to remove based on the deoptimization reason. First we profiled the benchmarks and counted the number of times each type of deoptimization was triggered in each benchmark. Then we modified the code generation phase to skip insertion of the conditional branch for the deoptimization reasons that were never triggered by a specific benchmark.

By executing the same benchmark multiple times we determined exactly when the deoptimization events did and did not occur. When the profiling pass showed that a specific type of check never triggered a deoptimization event, then it was safe to remove that type of check. Our technique could not be used to reduce the number of deoptimization checks for real applications because the checks we removed may have been needed in the case of different inputs, but it was safe for our benchmarks running with a known input set.

Unfortunately, if a specific deoptimization reason triggered even a single deoptimization event, we still needed to keep the check in order to ensure correct benchmark execution.[2] However, despite this restriction we were able to test with removing a majority of the checks for most of the benchmarks, and for some benchmarks only a negligible number of checks remained.

Figure 3 shows the number of checks that were removed and that remained for each of the benchmarks we evaluated. Codeload and zlib had a negligible number of checks to begin with, so only a negligible number could be removed. Typescript had a significant number of checks, but we were only able to remove a negligible number of them. As a result we saw no difference in performance between the baseline and when checks were removed, and so we do not include results from these benchmarks in the rest of our performance analysis. Pdfjs also had a significant number of checks, but we were only able to remove about 40% of the checks. We chose not to include results from this benchmark because we do not think we were able to properly characterize the performance overhead when less than half of the checks were removed.

In the rest of this section we refer to results from the unmodified V8 code as *baseline* and results when we skipped insertion of the conditional branches for deoptimization checks as *skip*.

B. Experiment Setup

To evaluate the performance overhead of deoptimization checks we ran our baseline configuration and our skip configuration on a variety of different x64 systems. After our initial characterization we selected four systems with very different microarchitectures to use for detailed analysis. The first two systems use Intel CPUs released in 2013 and fabricated in 22nm technology. One of them is a high performance system

[2]We also experimented with removing required checks, but this crashed the program. Results shown in this paper include all required checks.

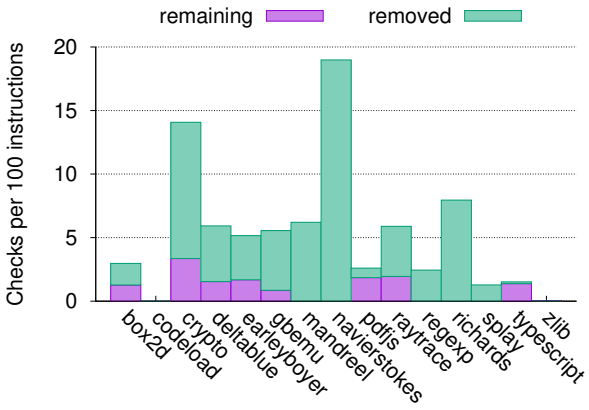

Fig. 3. Frequency of deoptimization checks that we were able to remove in our skip configuration, and frequency of checks that remained for each benchmark. Codeload and Zlib had a negligible number of checks to begin with and we removed less than half of the checks for Typescript and Pdfjs. We excluded results from these benchmarks from the rest of our performance analysis. For the remaining benchmarks we were able to remove a significant fraction of the total number of checks.

that uses the Haswell microarchitecture, and we refer to this as Intel Big Core (BC) in the rest of the paper. The other has a low power CPU that uses the Silvermont microarchitecture, and we refer to it as Intel Little Core (LC) in the rest of the paper. The other two systems used AMD processors released in late 2010 and early 2011, and they are also split between big and little cores. The detailed specifications for all systems are shown in Table I.

TABLE I
CPU SPECIFICATIONS

	Intel Big Core (BC)	Intel Little Core (LC)
Model	Xeon CPU E3-1275 v3	Celeron CPU N2820
Micro arch.	Haswell	Silvermont
Freq.	3.5 GHz	2.13 GHz
L1 i-cache	32 KB	32 KB
L1 d-cache	32 KB	24 KB
L2 cache	256 KB	1024 KB
L3 cache	8192 KB	–
	AMD Big Core (BC)	AMD Little Core (LC)
Model	Opteron Processor 6172	E-350
Micro arch.	Magny-Cours	Bobcat
Freq.	2.1 GHz	1.6 GHz
L1 i-cache	64 KB	32 KB
L1 d-cache	64 KB	32 KB
L2 cache	512 KB	512 KB
L3 cache	5118 KB	–

We ran all experiments using version 5.1.281.27 of the V8 JavaScript engine on systems running Arch Linux. Our baseline results were collected using an unmodified version of V8, while the skip checks results were only modified to skip insertion of a conditional branch if we had determined that the deoptimization reason associated with the check was never triggered for that benchmark. We set the "doDeterministic" flag in the Octane benchmarks to true to ensure that each benchmark ran for a fixed number of iterations rather than a fixed amount of time.

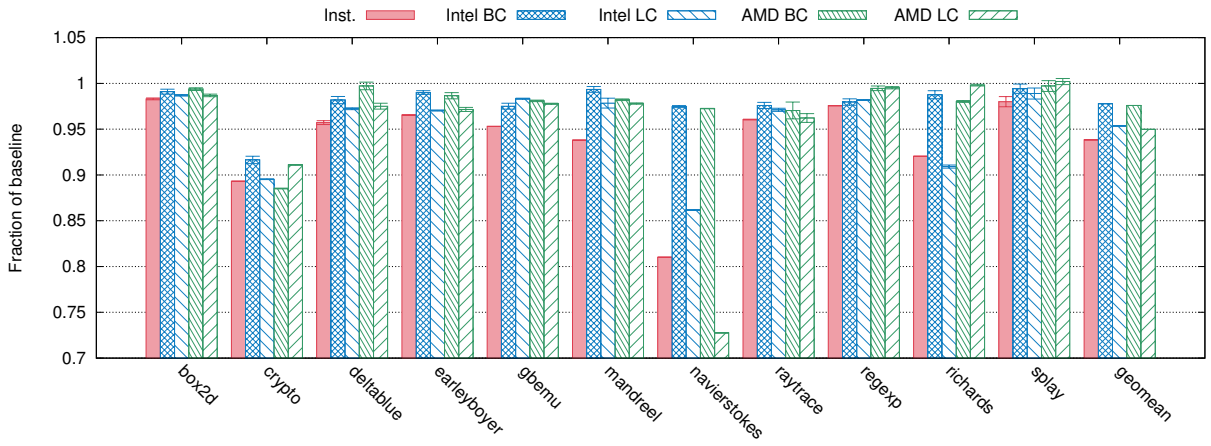

Fig. 4. Relative instruction count and execution time of the configuration that skips generation of conditional branches for deoptimization checks as a fraction to the unmodified V8 baseline results. The variation in instruction count was negligible between the four different test systems and so is shown only once and is the same for all of them. Execution time did vary significantly and the results show the relative execution time of the skip configuration on each system compared to the baseline execution time on the same system. In all cases lower is better since it indicates fewer instructions or faster execution time.

When running the benchmarks we made sure that the test systems were not running any other compute intensive workloads. However, there was still some variation in execution time as a result of the complex interaction between hardware, OS, and the V8 runtime system that uses JIT compilation and garbage collection. To account for this variation we ran each benchmark configuration 500 times. We report results based on the mean of the 500 benchmark executions for each measured statistic, and we also calculated the confidence interval at a 95% confidence level. We report the confidence interval in our graphs, although in most cases its bound is small enough that it is not easily visible on the graph. We also show the geometric mean as a summary of all the benchmark results for each metric that we report, but we do not include a confidence interval when reporting this summarized result. We recorded the execution time of each experiment and also used the Linux perf tool to collect performance counter results for instructions, cycles, branches, and branch misses.

C. Relative Performance

Earlier we showed how many checks we skipped for each benchmark (see Figure 3). Here we show how the reduction in instructions executed due to skipping the checks impacts the execution time of the benchmark. Figure 4 compares the relative instruction count and execution time of the baseline and skip results. The relative instruction count was very similar across all four systems so we only show it a single time.[3] But the relative execution time had significant variation between the different systems. For example, removing 8 checks per 100 instructions in Richards resulted in an 8% reduction in instruction count for both Intel BC and Intel LC. But on the Intel BC this only resulted in a 1.2% decrease in execution time, while on the Intel LC it resulted in a 9.1% decrease.

[3]For most benchmarks the instruction count was very stable between runs; only splay showed significant variation. Splay is designed to stress the garbage collection so this behavior is not surprising.

The general trend is that the systems with big cores tend to benefit less from removing checks than ones with little cores. On average across all of the benchmarks the instruction count was reduced by 7.2%. The Intel BC execution time was reduced by 2.2%, while the LC time dropped by 4.6%. Looking at the AMD CPUs, BC execution time dropped by 2.6% and the LC time dropped by 5.0%. However, for the AMD cores these results are skewed somewhat by Navierstokes, which has the largest reduction in execution time (nearly 30%) of any configuration tested.

Although the general trend is for the big cores to benefit less from removing checks, the extent varies significantly depending on the benchmark. For instance, Navierstokes is the benchmark where we removed the most checks, and the instruction count dropped by 18%. This resulted in only a 3.5% reduction in execution time on the Intel BC but a 14.8% reduction in execution time on the Intel LC. Crypto is another benchmark where we were able to remove a large fraction of checks, and the instruction count dropped by 11.7%. This corresponded to a 9.4% reduction in execution time for Intel BC and a 10.8% reduction for the Intel LC. To better understand this performance difference we looked at IPC and branch prediction accuracy.

D. IPC

Figure 5 shows the absolute IPC of the Intel BC and LC for each of the benchmarks for both the baseline and skip checks settings, and Figure 6 shows similar data for the AMD systems. In all cases we counted the total instructions and cycles that were actually executed by the benchmark for each of the different configurations and used this data to calculate IPC. This allowed us to compare the true instruction throughput for each configuration. But it also meant that higher IPC for the base configuration compared to the skip configuration didn't result in speedup, because the base configuration executed more instructions.

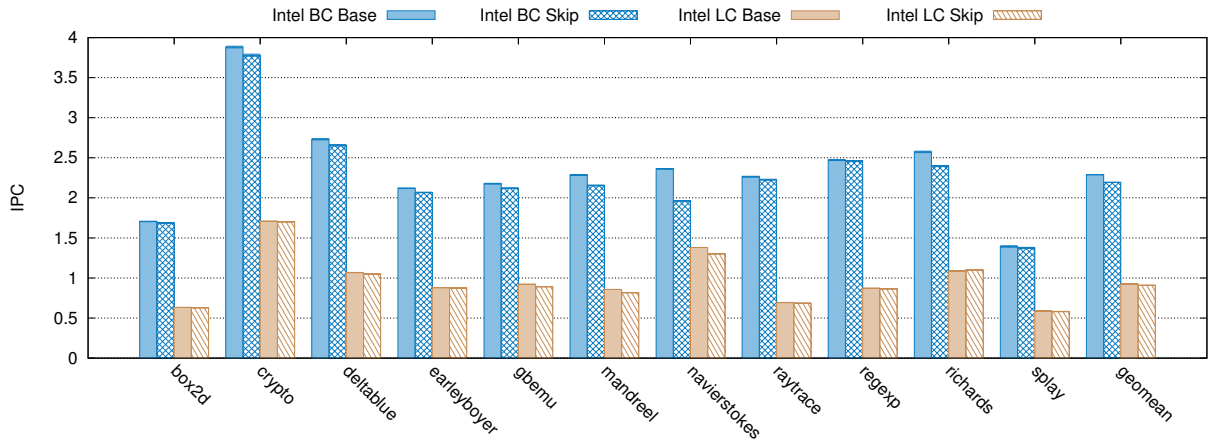

Fig. 5. Absolute IPC of the Intel Big Core (BC) and Little Core (LC) systems for both the baseline and skip checks configuration. The IPC of the BC is much higher than the LC, and the BC's IPC tends to drop in the skip configuration compared to the baseline, while the LC's IPC stays constant or decreases only slightly for most benchmarks.

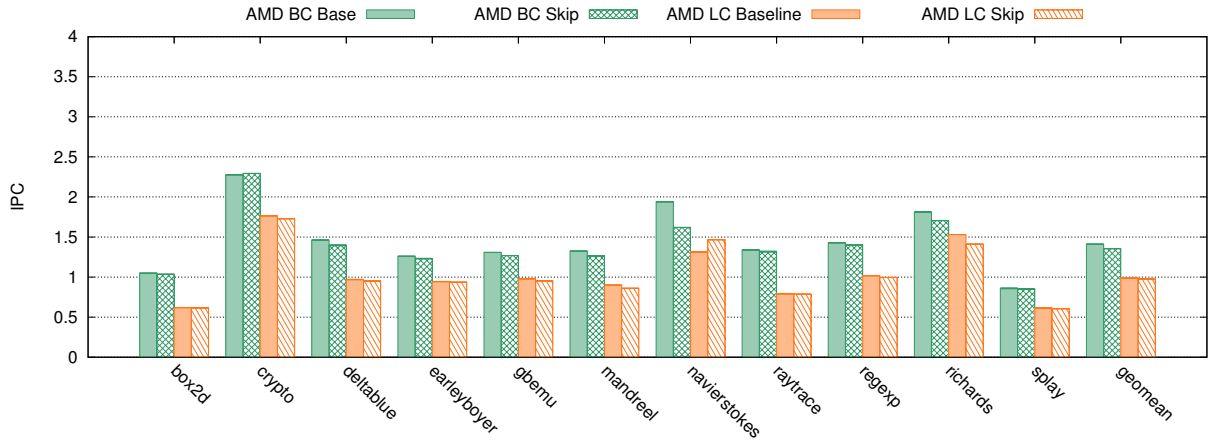

Fig. 6. Absolute IPC of the AMD Big Core (BC) and Little Core (LC) system for both the baseline and skip checks configuration. There is less variation than with the Intel systems. Still, the overall average shows a larger IPC drop for the BC system than the LC, although part of this is due to the results from Navierstokes where the IPC of the skip configuration actually increases for the LC.

The maximum issue width of the Intel BC is 4 and for the Intel LC it is 2, so the large absolute difference between the Intel BC and LC makes sense. However, the Intel BC has a noticeable drop in IPC for many of the benchmarks while the Intel LC has a negligible drop in IPC in most cases. This explains why the Intel BC shows less performance improvement on average than the Intel LC when skip checks configuration removes conditional branches. The branches that are removed are typically easy to predict, and they do not cause many stalls on the wide issue Intel BC. However, on the narrow issue Intel LC even these easily predicted branches reduce instruction throughput. As a result the LC benefits much more from removing the checks than the BC does.

However, for a benchmark like Crypto the Intel BC is executing at near its peak IPC of 4. So in this case even the easily predicted branches associated with the deoptimization checks contribute to limiting execution throughput. Removing these branches provides comparatively larger benefits for this benchmark than one like Navierstokes where peak throughput is not constrained by issue-width.

For the AMD systems the trends are similar, but Navierstokes is an interesting outlier where the IPC improves for the AMD LC when going from the baseline to skip checks configuration. This is somewhat surprising since the branches removed by the skip checks are easily predicted. In fact when looking at the branch prediction data the number of misses is not significantly reduced. However, the number of branches as a percentage of total instructions drops from 27% of instructions to 8% of instructions. It's unclear whether there is a limit on how many branches the AMD LC system can retire per cycle, or if it is hitting some other bottleneck that limits its throughput. The 27% of instructions as branches for the baseline is similar to many of the other benchmarks, but none of the other benchmarks with a high percentage of

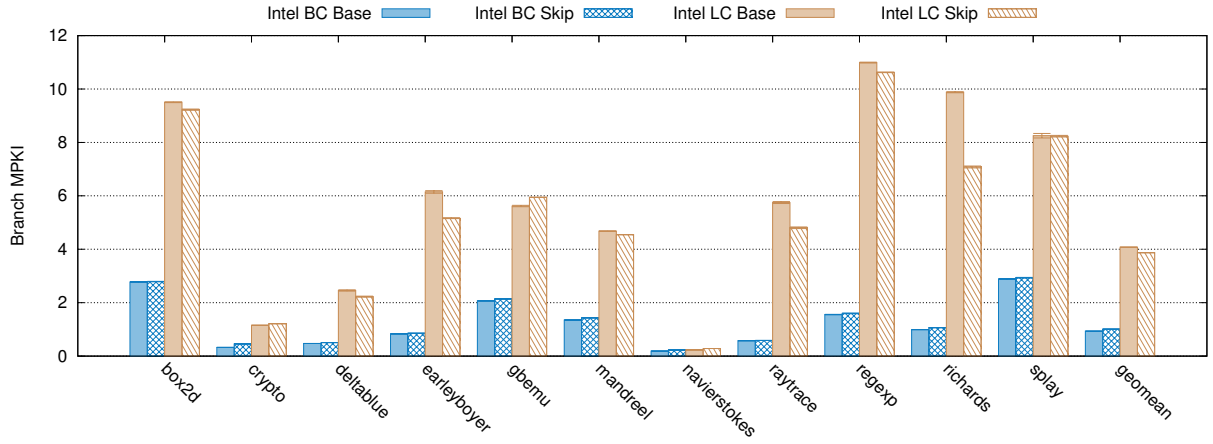

Fig. 7. Branch misses per thousand instructions (MPKI) for the Intel Big Core (BC) and Little Core (LC) systems. For most benchmarks the BC has much better prediction accuracy than the LC. As a result, removing branches in the skip configuration does not reduce the absolute number of misses for the BC, and the MPKI increases slightly as misses stay constant and the number of instructions decreases. For the LC, removing the conditional branches helps the overall prediction accuracy despite the fact that the branches are typically very easy to predict.

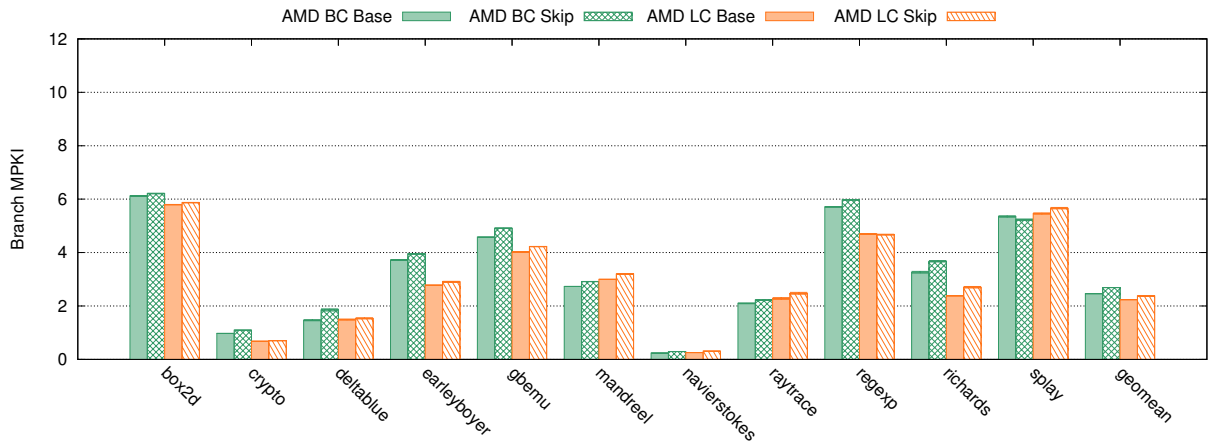

Fig. 8. Branch misses per thousand instructions (MPKI) for the AMD Big Core (BC) and Little Core (LC) systems. The prediction accuracy for both cores is better than the Intel LC, but worse than the Intel BC. Both of the AMD cores have MPKI increase slightly when removing the easily predicted branches used by the deoptimization checks.

branches have as dramatic a drop in branch count to bring the total below 10%.

E. Branch Prediction

The final set of data that we show is branch misses per thousand instructions (MPKI). Figure 7 shows results for the Intel systems and Figure 8 shows results for the AMD systems.

In the case of the Intel BC the MPKI always increases when going from the baseline configuration to the skip configuration. In most cases this is because the total number of instructions is reduced, while the number of branch misses stays constant. However, Crypto was interesting because the absolute number of branch mispredictions increased when going from the baseline to the skip configuration. Removing the easily predicted branches in the baseline configuration apparently contributed to increasing the total number of mispredictions, not just the misprediction rate. Regardless, Crypto was still the benchmark that benefited the most from removing branches on the Intel BC.

The Intel LC shows opposite behavior to that of the BC. For the LC the MPKI is reduced from most benchmarks when switching from the baseline to skip configuration. Typically the branch prediction accuracy is still better for the baseline configuration because most of the branches that are removed are easily predicted, but even these easily predicted branches have some mispredictions, and so removing them reduces the overall number of mispredictions per instruction.

The results for the AMD systems were mixed, as there was not as great a difference between the BC and LC as there was for the Intel systems. The overall MPKI for both AMD systems is worse than the Intel BC, but better than the Intel LC. The difference between the AMD BC and LC is less pronounced

than it is for the Intel systems.

V. Related Work

JavaScript performance has been widely studied and a variety of hardware and software optimizations have been proposed to improve the performance of optimized code or reduce the overhead associated with deoptimization checks.

The most closely related work is from Dot et al. [8]. Like us they are interested in analyzing performance overheads associated with dynamic languages. They also proposed ISA extensions to reduce the overhead of deoptimization checks and evaluated the proposed extensions using a combination of data collected with Pin and simulation results from the Sniper simulator.

Like them we were interested in the potential for hardware optimization to reduce the overheads imposed by dynamic languages. But we chose a different technique to evaluate the potential benefit of possible hardware optimizations. Our technique involved measuring performance using real systems while modifying the code generated by the V8 JIT compiler. We believe this allowed us to get better insights into the true performance impact of the conditional branches used by deoptimization checks. By using real systems we were able to run the benchmarks to completion multiple times and collect statistically significant data even in the presence of system noise associated with the complex interaction of a runtime system that includes a two-tiered JIT compiler and uses garbage collection for memory management, all running on a full Linux OS. We also evaluated four systems with very different microarchitectures rather than a single design point. Our work is more of a limit study on the potential benefits of reducing the number of branches associated with deoptimization checks. Conversely, their work provides an evaluation of a specific technique that could be used to achieve this goal and is only simulated for one specific design point.

Anderson et al. [3] also observed that dynamic languages have a large number of runtime checks and proposed hardware extensions intended for mobile processors to decrease the overhead of checking that loads had the correct type. Their performance analysis focused on simpler in-order cores that were common in mobile processors at the time. In contrast, our work focuses on out-of-order designs that are more common today in both server systems and high end mobile CPUs.

Several researchers have studied JavaScript code in an effort to characterize it and understand the primary performance bottlenecks and their impact on the microarchitecture of contemporary CPUs.

Zhu et al. [21] studied the performance of event driven applications that run with node.js. Although node.js uses the V8 JavaScript engine to execute JavaScript code, their work focused on characteristics specific to the event driven programming model, rather than overheads in the V8 JavaScript engine associated with running optimized JavaScript code.

Ogasawara [14] also studied the performance characteristics of node.js workloads. He showed that most of the execution time is spent in C++ library calls, rather than running JavaScript code that can be optimized by the V8 runtime. This contrasts with our analysis since we focused on understanding overheads associated with the optimized code generated by the V8 JIT compiler.

Musleh and Pai [13] studied the impact of a variety of microarchitectural parameters on the V8 JavaScript engine, but they did not look specifically at the overhead of deoptimization checks, which is what we focused on.

Auler et al. [4] analyzed how possible JavaScript compiler optimizations can have different performance impacts depending on the system executing the code. They also proposed a methodology to automatically crowdsource the optimal configuration for a specific device.

Ahn et al. [2] proposed modifications to how V8 constructs types and demonstrated performance improvements on the JSBench benchmarks. Their work differs from ours because their proposed modifications were intended to increase the amount of time that V8 spent executing optimized code, whereas our work focused on deoptimization checks overhead that are only present when executing optimized code.

Kedlaya et al. [10] observed that when managed language runtimes (such as the JVM or CLR) are used to run dynamic languages, they typically do not include support for deoptimizations. This limits the quality of code that can be generated and inhibits common optimizations. They propose a way to do this and implemented it in MCJS, a JavaScript engine implemented on top of Microsoft's CLR runtime.

There has also been some work characterizing the behavior of JavaScript benchmarks in an effort to understand how they compare to real-world JavaScript code commonly seen on websites:

Richards et al. [19] performed a detailed analysis of how JavaScript is used in practice. They found many examples of widely used techniques that inhibit optimization. Their work is complementary to ours since it suggests the need for changes to increase the fraction of time that JavaScript engines can spend running optimized code. In contrast, we characterized overheads that only exist for optimized code.

Ratanworabhan et al. [18] performed a similar study and compared the characteristics of 11 commonly used websites with those of the SunSpider and V8 benchmarks (note that the V8 benchmark suite is a predecessor to the Octane benchmark suite, and is not the same as the V8 JavaScript engine).

Tiwari and Solihin [20] also studied benchmark characteristics, although in their case they compared the V8 benchmark suite with SunSpider. Like us, they used the V8 JavaScript engine and collected hardware performance counter results while running on real machines. Unlike us, however, they did not look specifically at deoptimization checks; instead they focused on analyzing the similarity of benchmarks in the V8 and SunSpider suites.

VI. Conclusion

Type speculation is a proven and effective technique for improving the performance of compute intensive code written in dynamic languages. But speculation can be incorrect, and the runtime system needs to detect when this happens and revert to running unoptimized code. The process of reverting from optimized code back to unoptimized code is called deoptimization, and the optimizing compiler inserts many

deoptimization checks in the code that it generates. These checks consist of a comparison, or for overflow checks a condition flag that may be generated implicitly by a preceding arithmetic operation, and a conditional branch.

We evaluated the frequency of the deoptimization checks in code generated by the V8 JavaScript engine running the Octane benchmarks. We then evaluated the performance overhead associated with the conditional branch when running the complete benchmarks on four different real systems with varying microarchitectural parameters. This was done by skipping the branch insertion during the code generation phase in cases where we knew the benchmark would never trigger the branch.

The performance overhead of the deoptimization check's conditional branches varied significantly depending on the system parameters. On average we eliminated 6.2% of instruction by skipping the branches. This resulted in only a 2.2% performance improvement on our system with a high performance Intel CPU, but on an Intel CPU targeted towards low power operation performance improved by 4.6%.

The reason for this difference is that the high performance system had a wider issue width and a more accurate branch predictor that was capable of predicting more branches per cycle. The branches associated with deoptimization checks are highly biased towards not taken, and as a result they are easily predicted.

Previous work has suggested ways to reduce the overhead of deoptimization checks by eliminating the conditional branch and using special instructions that perform checks implicitly and trigger an exception in the rare case where the check fails. However, our analysis shows that even when checks are frequent, their performance overhead may be limited on wide issue CPUs. The easily predicted branches may not be a bottleneck for system throughput.

Wider issue CPU cores inherently consume more power, and current trends are for CPUs to have both high performance "big" cores along with simpler power optimized "little" cores. Our analysis indicates that little cores may benefit more than big cores from optimization to reduce the number of branches for deoptimization checks. In addition our analysis only considered the impact of removing conditional branches from the code generation phase. It may be possible to achieve additional benefits by making changes earlier in the code generation pipeline, but this would depend on the specific optimizations proposed and their interaction with the language runtime system.

Overall our study characterizes the overhead of conditional deoptimization checks in a state-of-the-art dynamic language runtime system running on modern out-of-order processors with full OS and language runtime interaction. This complements previous work that has studied other aspects of these workloads using various architectural simulators.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their feedback on the paper. This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] "Octane benchmarks." [Online]. Available: https://developers.google.com/octane/

[2] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving javascript performance by deconstructing the type system," in *PLDI 2014*, Jun. 2014.

[3] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked load: Architectural support for javascript type-checking on mobile processors," in *HPCA 2011*, Feb. 2011.

[4] R. Auler, E. Borin, P. de Halleux, M. Moskal, and N. Tillmann, "Addressing javascript jit engines performance quirks: A crowdsourced adaptive compiler," in *CC 2014*, Apr. 2014.

[5] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 2, 1995.

[6] J.-D. Dalton, G. Seth, and L. Lafreniere, "Announcing key advances to javascript performance in windows 10 technical preview," Oct. 2014. [Online]. Available: http://blogs.msdn.com/b/ie/archive/2014/10/09/announcing-key-advances-to-javascript-performance-in-windows-10-technical-preview.aspx

[7] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *POPL 1984*, Jan. 1984.

[8] G. Dot, A. Martnez, and A. Gonzlez, "Analysis and optimization of engines for dynamically typed languages," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2015.

[9] U. Hölzle, "Adaptive optimization for self: reconciling high performance with exploratory programming," Ph.D. dissertation, Stanford University, 1995.

[10] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf, "Deoptimization for dynamic language jits on typed, stack-based virtual machines," in *VEE 2014*, Mar. 2014.

[11] R. Loui, "In praise of scripting: Real programming pragmatism," *Computer*, vol. 41, no. 7, Jul. 2008.

[12] K. Millikin and F. Schneider, "A new crankshaft for v8," Dec. 2010. [Online]. Available: http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html

[13] M. Musleh and V. Pai, "Architectural characterization of client-side javascript workloads and analysis of software optimizations," Purdue University, Tech. Rep., 2015.

[14] T. Ogasawara, "Workload characterization of server-side javascript," in *IISWC 2014*, Oct. 2014.

[15] J. Ousterhout, "Scripting: higher level programming for the 21st century," *Computer*, vol. 31, no. 3, Mar. 1998.

[16] N. Pierron, "Ionmonkey: Optimizing away," Jul. 2014. [Online]. Available: https://blog.mozilla.org/javascript/2014/07/15/ionmonkey-optimizing-away/

[17] F. Pizlo, "Introducing the webkit ftl jit," May 2014. [Online]. Available: https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/

[18] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, Jun. 2010.

[19] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *PLDI 2010*, Jun. 2010.

[20] D. Tiwari and Y. Solihin, "Architectural characterization and similarity analysis of sunspider and google's v8 javascript benchmarks," in *ISPASS 2012*, Apr. 2012.

[21] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in *MICRO 2015*, Dec. 2015.