Server-Side Type Profiling for Optimizing Client-Side JavaScript Engines

Madhukar N. Kedlaya
University of California, Santa Barbara, USA
mkedlaya@cs.ucsb.edu

Behnam Robatmili
Qualcomm Research Silicon Valley, USA
behnamr@qti.qualcomm.com

Ben Hardekopf
University of California, Santa Barbara, USA
benh@cs.ucsb.edu

Abstract

Modern JavaScript engines optimize hot functions using a JIT compiler along with type information gathered by an online profiler. However, the profiler's information can be unsound and when unexpected types are encountered the engine must recover using an expensive mechanism called *deoptimization*. In this paper we describe a method to significantly reduce the number of deoptimizations observed by client-side JavaScript engines by using ahead-of-time profiling on the server-side. Unlike previous work on ahead-of-time profiling for statically-typed languages such as Java [13, 24] our technique must operate on a dynamically-typed language, which significantly changes the required insights and methods to make the technique effective. We implement our proposed technique using the SpiderMonkey JavaScript engine, and we evaluate our implementation using three different kinds of benchmarks: the industry-standard Octane benchmark suite, a set of JavaScript physics engines, and a set of real-world websites from the Membench50 benchmark suite. We show that using ahead-of-time profiling provides significant performance benefits over the baseline vanilla SpiderMonkey engine.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Run-time environments, Optimization

Keywords Deoptimization, JavaScript, profiling, virtual machine, language implementation

1. Introduction

JavaScript has emerged as the de facto language of the web. With websites becoming more and more JavaScript-heavy, JavaScript performance has become an ever greater concern. Modern JavaScript engines have multi-tier execution architectures with sophisticated optimizing JIT compilers. Like optimizing JIT compilers for statically-typed languages (e.g., the JVM [3] and CLR [2]), JavaScript JIT compilers optimize based on profile information collected during execution. But unlike those other JITs, the collected profile information for JavaScript is of a different nature involving heuristic type information that is not guaranteed to be correct. When a function is optimized using profile-based type assumptions, there is a chance that those assumptions will not hold in the future. The JavaScript JIT compiler will optimize a hot function based on the types observed during the previous executions of the function. In the future, if new, unexpected types are encountered during execution of the optimized code, the JavaScript engine must employ a recovery mechanism called *deoptimization* to guarantee correctness. This recovery mechanism is a heavy-weight, expensive process that can severely impede the engine's performance.

In this paper we propose a technique that uses ahead-of-time type profiling on the webserver side in order to determine type and hotness information for a JavaScript program; that information is sent to the web browser client as commented annotations in the JavaScript code, and the client uses that information to reduce the number of deoptimizations during execution. Client JavaScript engines that are aware of the ahead-of-time profiling information can take advantage of it, while client engines that are not aware of it can safely ignore it. The intent of this technique is *not* to reduce profiling or compilation overhead (which turn out to be mostly insignificant), but rather to reduce the number of deoptimizations during program execution and also to enable more aggressive and earlier optimization of functions without having to fear increased deoptimizations.

A naïve approach to ahead-of-time type profiling for JavaScript would simply observe the execution of the program on some set of inputs and (1) mark all functions that become hot sometime during the execution, so that they can

be optimized immediately instead of waiting; and (2) remember all types seen during the execution of those hot functions, so that the optimized versions will not have to be deoptimized due to type changes. However, it turns out that this naïve approach would significantly *degrade* performance on the client and would also create program annotations potentially orders of magnitude larger than necessary. We explain the reasons behind this observation and our key insights that allow ahead-of-time profiling to be both practical and effective.

Previous work for statically-typed language JIT compilers has proposed using ahead-of-time profiling, as discussed in Section 2. However, JavaScript provides a new setting that requires new techniques and insights. We show that for JavaScript: deoptimization is an important performance concern; ahead-of-time profiling can provide significant performance benefits by avoiding deoptimization; and the annotation comments in the JavaScript code sent from the server increase code size by only a small fraction. The specific contributions of this work are:

- We describe a method for ahead-of-time profiling of JavaScript programs to collect type and hotness information. We identify the key kinds of information and places to collect that information that provides the most benefit for optimization without requiring excessive annotations on the program code being sent over the network.
- We describe a method for JavaScript engines to take advantage of the ahead-of-time profiling information to reduce deoptimizations and to more aggressively optimize functions without incurring increased deoptimizations.
- We evaluate our ideas using Mozilla's JavaScript engine SpiderMonkey. Our experiments show that our technique is beneficial for both load-time and long-running JavaScript applications, as represented by the Membench50 load-time benchmark suite, the industry-standard Octane performance benchmark suite, and a set of open-source JavaScript physics engines. We measure the performance using three different criteria: execution time for Octane benchmarks, frames per second (FPS) for the JavaScript physics engines, and reductions in number of deoptimizations for the Membench50 benchmarks. Our evaluation shows a maximum speedup of 29% and an average speedup of 13.1% for Octane benchmarks, a maximum improvement of 7.5% and an average improvement of 6.75% in the FPS values for JavaScript physics engines, and an average 33.04% reduction in deoptimizations for the Membench50 benchmarks.

The rest of the paper is organized as follows. Section 2 describes related work on optimizing JIT compilers. Section 3 provides background information on the JavaScript language and on modern JavaScript engine architectures. Section 4 describes the concepts behind our technique. Section 5 describes our evaluation methodology and results, and Section 6 concludes.

2. Related Work

Our work builds on decades of research into optimizing JIT compilers, such as Self [20, 22], Java HotSpot VM [27], Jalepeno [12], PyPy [15], Google's V8 engine [30], Mozilla's SpiderMonkey [28], and WebKit's JavaScriptCore [23]. We review the most relevant of that related work below.

2.1 Ahead-of-Time Profiling

Ahead-of-time profiling for the purpose of optimizing a JIT compiler is not a new idea, but previous efforts have focused on statically-typed languages such as Java and C#. JavaScript, a dynamically-typed language, provides a new setting that dramatically changes the required insights and techniques. In particular, the most important optimization performed by a JavaScript JIT compiler is type specialization based on unsound heuristics such as online type profiling. Because type specialization is unsound, the engine must be able to deoptimize the specialized code when it encounters unanticipated types. Deoptimization is an important cause of performance loss and is the main target of our technique, unlike any of the previous ahead-of-time profiling techniques described below. We do in addition follow previous work in using ahead-of-time profiling to detect hot functions that can then be compiled early. However, our new setting also influences this existing technique in new ways because merely detecting *hotness* is insufficient—we must also ensure that the hot function is *type stable* for early compilation to have any benefit, otherwise deoptimization is likely to happen. We now describe the previous work on ahead-of-time profiling for JIT compilation, which all target statically-typed languages.

Krintz and Calder [24, 25] describe an approach to identify hot functions and hot callsites in Java programs using analysis information collected offline. This information is used by the JIT compiler to guide its optimization heuristic. Our approach is similar to their approach of using offline data to guide online optimizations. Unlike their approach, our offline profiler collects type information and deoptimization information in addition to hot functions and hot callsite information. The type information is important for a dynamically-typed language such as JavaScript because most of the optimizations that are performed in the optimizing compiler depend on stable type information. Deoptimization information helps to figure out possible places where deoptimizations occur in the hot functions and the reasons why deoptimization has happened. This information helps the optimizing compiler to make better decisions while compiling those hot functions.

Arnold et al. [13] describe an Java virtual machine architecture that uses a cross-run profile repository to improve performance. The main idea described in that paper is to capture the profile data at the end of the execution of the program instead of discarding it after every run. This collective profile information is used to guide the selective optimization of functions based on metrics like future use. A key idea of

that work is to address the *compilation time* vs. *future execution time* trade-off inherent in single-threaded engines that interleave execution and JIT compilation. Modern JavaScript engines employ concurrent JIT compilation, and so compilation time is generally not as important an issue. Also, we take advantage of the client/server infrastructure inherent in the world-wide web to do the ahead-of-time profiling on the server side and send the resulting information to the client for it to take advantage of, rather than doing profiling in the client itself.

2.2 Type Annotations for JavaScript

Developers and researchers have created several typed variants of JavaScript. These variants are either restricted subsets of the full JavaScript language or do not allow the types to be used by the JIT compiler for optimization.

The JavaScript dialect `asm.js` [14] is a strict subset of JavaScript that is intended to be generated by compilation to JavaScript from some statically-typed language such as C. It indicates the types of variables and operations based on subtle syntactic hints and a `"use asm"` prologue directive. Though this enables the JavaScript engine to perform ahead-of-time compilation and faster execution, the `asm.js` syntax is very restrictive and is not suitable for writing modern webpages. It is designed to be an intermediate representation for porting applications written in statically-typed languages into the web browser. In contrast, our approach deals with already existing JavaScript programs and handles the entire JavaScript language.

Flow [18] is a static type checker for JavaScript that allows type annotations in the syntax. These annotations are used by the compiler to type-check the code for correctness. An optimizing JIT compiler cannot make use of these annotations because the annotations are erased during the translation of Flow code to JavaScript. This is also true of Google's Closure compiler [16], which allows type annotations in the JavaScript code, and of TypeScript [29], a typed superset of JavaScript.

2.3 JavaScript Engine Optimizations

Guckert et al. [19] show that persistent caching of compiled JavaScript code across visits to the same webpage helps reduce compilation time by up to 94% in some cases. However, because the optimizing compiler usually runs in a separate parallel thread compilation time is not much of a concern in modern JavaScript engines. In addition, this technique does not translate well to a setting where the server is responsible for collecting information and sending it over the network to clients, due to the large size of the compiled code and its specificity to a particular architecture.

Oh and Moon [26] describe another client-side optimization technique targeting load-time JavaScript code (i.e., JavaScript code executed when a webpage is loaded by the browser). This technique caches snapshots of the heap objects that are generated during the load time; the snapshots are serialized during caching and then deserialized when the page is reloaded. This approach uses significant amounts of storage space and does not translate well to server-side profiling.

3. Background

In this section we describe background on the JavaScript language and modern JavaScript engines required in order to understand the key concepts discussed in this paper.

3.1 The JavaScript Language

JavaScript is an imperative, dynamically-typed scripting language with objects, prototype-based inheritance, higher-order functions, and exceptions. Objects are the fundamental data structure in the language. Object properties (the JavaScript name for object fields) are arbitrary strings and can be dynamically inserted into and deleted from objects during execution. Because property names are just strings, a JavaScript program can compute a string value during execution and use it as a property name in order to access an object's existing property or to insert a new property. A form of runtime reflection can be used for object introspection in order to iterate over the properties currently held in an object. Even functions and arrays are just different kinds of objects, and can be treated in the same way as other objects, e.g., inserting and deleting arbitrary properties. JavaScript is designed to be resilient even in the face of nonsensical actions such as accessing a property of a non-object (i.e., a primitive value) or adding two functions together; such cases are handled using implicit type conversions and default behaviors in order to continue execution as much as possible without raising an exception.

3.2 JavaScript Engine Architecture

Modern JavaScript engines rely heavily on profiling and JIT compilation for performance. The JIT compiler relies on type information gathered by the profiler in order to enable effective optimizations in the face of JavaScript's inherent dynamism. Type information includes not just the primitive kinds of values (`number`, `boolean`, `string`, `object`, `undefined`, and `null`), but in addition information about *object shape*, i.e., a list of object properties and their offsets in the object. Because properties can be arbitrarily added to and deleted from an object, object shapes can change frequently during execution.

Figure 1 shows a typical multi-tier architecture for a generic JavaScript engine, based on the designs of existing production JavaScript engines such as Google's V8 [17], Mozilla's Spidermonkey [28], and WebKit's JavaScriptCore [23]. These tiers operate at the granularity of individual functions.

Tier 1. The first tier of execution is a fast interpreter for parser-generated bytecode. The interpreter is used to ensure quick response times during execution of the JavaScript program. For example, SpiderMonkey's bytecode interpreter

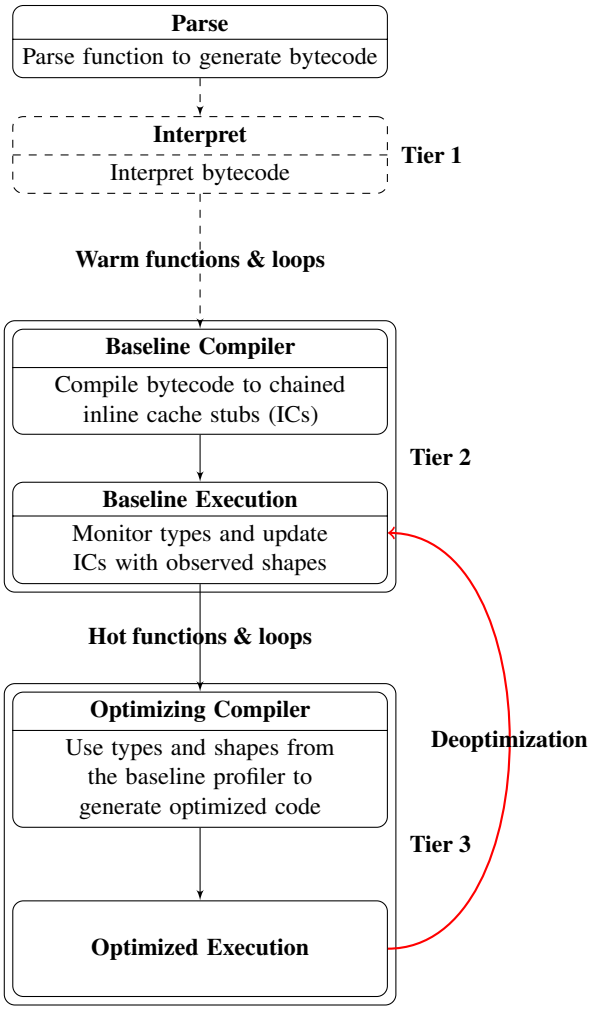


Figure 1: Flow graph showing different phases of execution in a generic JavaScript engine. The interpretation phase, represented by dashed lines, is an optional phase in JavaScript engines like Google's V8.

and JavaScriptCore's LLInt execute functions for the first 10 and 6 times they are called, respectively. Once the given threshold is reached, the function is considered warm. Not all engines use this first tier; for example, the V8 engine skips the interpreter and goes straight to tier 2.

Tier 2. The second tier of execution is a baseline compiler that compiles the bytecode to assembly code as quickly as possible, with minimal optimization. The baseline compiler also inserts instrumentation into the compiled code to collect profiling information. The profile information that is collected by the baseline compiler includes the types of variables and of object properties and the shapes of objects whose properties are accessed/modified during function execution. The baseline code is executed many times before a function is considered hot, e.g., SpiderMonkey typically executes the baseline compiled code one thousand times before moving to the next tier [1]. This large threshold is intended to help ensure that the type information gathered by the profiler is stable and hopefully will not change in the future (if it does change then *deoptimization* will be triggered, discussed in Section 3.3). This threshold may vary based on other factors such as whether a function contains back-edges which are frequently visited or whether a function has been deoptimized earlier.

Tier 3. The third tier of execution is an optimizing compiler that compiles hot functions based on profile information collected in the previous runs of the function, i.e., the baseline compiled code. The profile information may be invalidated by future calls to the function being optimized (e.g., the types may change), therefore the optimized code also contains *guards* that check the assumptions under which the code was compiled. If those guards are violated then deoptimization will happen, moving execution from the optimized code back to the baseline compiled code from tier 2. The optimizing compiler performs various optimizations such as loop invariant code motion, common subexpression elimination, guard hoisting, function inlining, and polymorphic cache inlining to speed up the execution of the function. The optimizing compiler is relatively slow compared to the interpreter and baseline compiler, therefore modern JavaScript engines adopt a concurrent compilation strategy. Using this strategy, the time taken for compilation is not a big concern for performance because it is not in the critical path for program execution.

3.3 Type Specialization and Deoptimization

Many of the most effective optimizations performed by the optimizing compiler are based on *type specialization*. For example, consider the expression "`a + b`". In the general case (without type specialization), variables `a` and `b` will refer to boxed values residing in the heap that are tagged to specify the types of those values; these are called *dynamic values*. To perform the + operation, the code must unbox `a` and `b`, determine their respective types based on their tags (requiring a series of branch instructions), perform any type conversions necessary, perform the operation, then box the resulting value along with its type tag. This process must be used for *all* operations on dynamic values, significantly slowing the execution time.

If, however, the compiler has reason to believe that `a` and `b` are (almost) always of certain types based on the profile information collected by the baseline compiled code, then it can specialize the optimized function to those types. It does not need to use dynamic values for `a` and `b`, it can use unboxed values instead; it does not need to check type tags to determine what operation to perform for +, it can directly use the operation pre-determined by the known types of `a` and `b`, and it can optimize the emitted code based on this knowledge. This is an example of type specialization, and it is one of the most effective means available for improving execution time of dynamically-typed languages. Another example of type specialization takes advantage of object shape information to

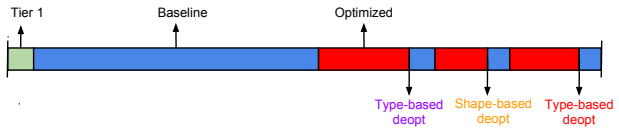


Figure 3: Timeline showing the execution of a single function within a JavaScript program in the server.

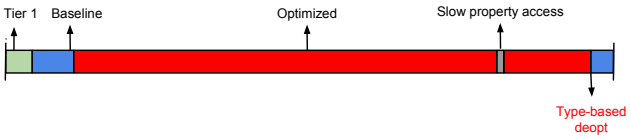


Figure 4: Timeline showing the execution of a single function within a JavaScript program in the client.

efficiently access object properties, e.g., using polymorphic inline caches [21].

The essential problem is that the profile information on types and object shapes is necessarily unsound—observed types during profiling do not guarantee what the types will be in future executions. Deoptimization is the recovery mechanism the engine uses when current types do not match the assumptions used when optimizing the function in tier 3. The engine does not discard the baseline compiled code from tier 2 when it generates the optimized code in tier 3. Instead, for each guard point where type information is checked and may be invalidated, the engine maintains a mapping from the optimized code to the equivalent point in the baseline compiled code. When a guard fails, execution stops at that guard in the optimized code and resumes at the equivalent point in the baseline compiled code, which is not type specialized and hence can handle any possible type. Deoptimization is an inherently expensive operation, and reducing the number of deoptimizations is a primary goal when optimizing engine performance.

Deoptimizations can be classified into different categories depending on their exact cause. A **type-based** deoptimization is caused by attempting to use a value that has a different primitive type than expected (`number`, `boolean`, `string`, `object`, `undefined`, or `null`). A **shape-based** deoptimization is caused by attempting to access an object that has a different shape than expected (i.e., the property offsets are potentially different). These are the two kinds of deoptimizations that we target in this paper.

Other kinds of deoptimizations are caused by speculative, optimistic assumptions made by the optimizing compiler that may not be valid. For example, because arrays are just objects in JavaScript, array elements in the middle of the array can be deleted leaving a "hole" in the array. The optimizing compiler assumes that there are no holes in an array. Numbers in JavaScript are doubles, but the optimizing compiler assumes that they are integers for added performance. Computed property accesses (i.e., computing an arbitrary string and using it as a property name) can be anything, but the optimizing compiler assumes that it will be a property actually in the object being accessed. For all of these assumptions the compiler must emit a guard to check that assumption in case it is not true, and trigger a deoptimization if it is not. We do not focus on these kinds of deoptimizations in this work, but they are an interesting target for future work.

4. Ahead-of-Time Type Profiling

In this section we describe our technique for performing ahead-of-time profiling on the server-side and for taking advantage of that profile information on the client-side. Figure 2 gives a high-level overview of our technique's flow. The server will profile the JavaScript program in two phases to collect profile information. This usually happens during the feature testing or regression testing phase of the application. Instead of a regular browser, the developers use a lightly modified version of the browser (as described later in this section) while performing the manual or automated regression testing, in order to collect the profile information. The application is then annotated with this profile information. The advantage of this approach is that whenever the application is updated, a new profile can be captured using the existing test suite and a new version of the annotated program can be created. The annotated version is then sent over the network to the client on request; the client can take advantage of that information if it is aware of the ahead-of-time profiling, or safely ignore that information if it is not.

We now describe in detail the server's ahead-of-time profiling technique and the insights required to collect the most useful information, and then the modifications required of the client to take advantage of the profile information. To clarify a potential point of confusion: we distinguish between the engine's standard runtime profiler used to collect information for the optimizing compiler (the *online profiler*) and our ahead-of-time profiler (the *offline profiler*) that collects the information gathered by the online profiler and preserves it past the end of the program's execution.

4.1 Server-Side Profiling

We concentrate our efforts specifically on type-based and shape-based deoptimizations. We currently ignore the other kinds of deoptimizations discussed in Section 3, though they may be interesting targets for future work. The ahead-of-time profiler operates in two phases: the **initial phase** and the **stability testing phase**. We begin with the initial phase.

4.1.1 Initial Profiling Phase

Consider Figure 3, which shows an example execution timeline for a single function within the program being profiled, where time is measured in number of function invocations. The function starts off in tier 1 (the green portion). Once the

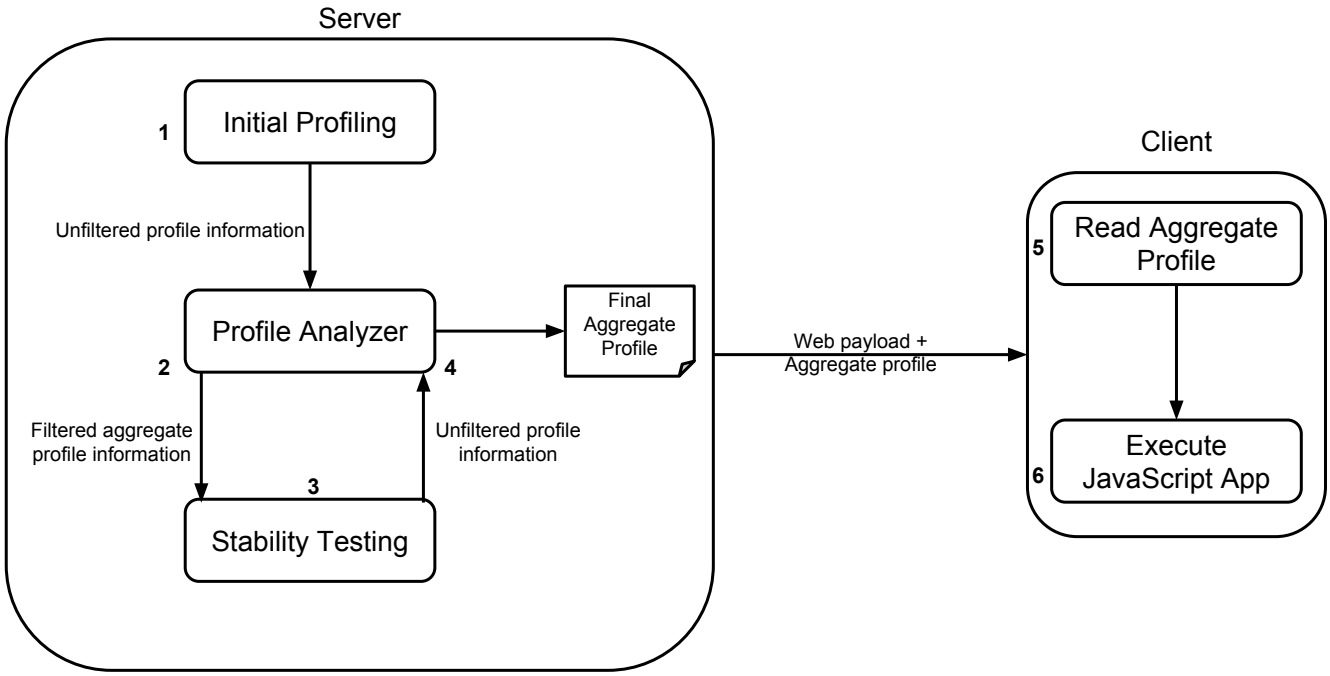


Figure 2: Profiling and execution setup. (1) The server performs initial profiling of the JavaScript application to generate unfiltered profile information. (2) The profile analyzer filters out relevant information and invokes (3) stability testing. (4) The profile analyzer collects the profile information from (3) and (1) to generate an aggregate profile. When the client requests the web application, aggregate profile information is sent along with the webpage. The client (5) reads the aggregate offline profile information and combines it with online profile information during the (6) execution of the application.

function becomes warm it goes through the baseline compiler and starts executing the baseline compiled code (the blue portion). During this time the online profiler is collecting information about types and object shapes. Once the function becomes hot it goes through the optimizing compiler and starts executing the optimized compiled code (the red portion). If the function must be deoptimized, for example, by a changing type or object shape, then the function reverts back to the baseline compiled code (and again is being profiled by the online profiler). If the function subsequently becomes hot again then it is recompiled by the optimizing compiler. The optimizing compiler may inline function calls as part of its optimizations, thus the optimized function may incorporate additional code over the baseline version.

By modifying the online profiler to save its information to an external file (a simple change to an API that all modern browsers already provide for accessing the online profile information), the offline profiler has access to each function's timeline, including whether the function ever became hot and all of the profile information about types and object shapes used by the function, as well as the number of deoptimizations that happened, where they happened, and why they happened.

Naïvely, one could attempt to optimize the program by marking each hot function and providing all of the collected type and shape information. The client could then immediately compile all hot functions (with parallel compilation to avoid load-time latency), using the provided type and shape information. Ideally this would allow the client to use optimized code right away while avoiding all (type- and shape-based) deoptimizations found during offline profiling (in Figure 3, all of the blue portions would be replaced by red portions). However, this approach does not work for three reasons.

Reason 1. Tracking shape information requires a lot of data, significantly increasing the amount of annotations that need to be sent over the network. Also, there is a large cost for the client in terms of memory and time, because the client needs to keep track of the executing program's object shape information in order to take advantage of the information from the ahead-of-time profiler. Our preliminary experiments show that tracking shape information in the client takes at least several megabytes of memory and increases client execution time by an average of 27%. We want to be able to reduce shape-based deoptimizations without incurring this overhead. Therefore, rather than have the offline profiler record actual shape information we instead have it record the program points at which shape-based deoptimization happened during the ahead-of-time profiled execution. This information is sent to the client instead of the exact shape

information; we explain how the client uses this program point information in the next subsection.

Reason 2. There is a very common coding idiom used by JavaScript programmers that uses the primitive JavaScript values `null` or `undefined` as a sentinel value to signal the end of some iteration. For example, think of a list of integers that is terminated by a `null` value to signal the end of the list. A variable `x` holding the value of the current position in the list would be an integer up until the point that sentinel value is reached, then variable `x` changes type to `null` instead. This type change then triggers a type-based deoptimization. We could eliminate that deoptimization by compiling the function at the beginning knowing that `x` could be an integer or `null`. However, the problem is that this newly optimized function would actually run *slower* than the original optimized function with the deoptimization, because the original compiled function could perform many optimizations relying on `x` being an integer and is thus much faster than the newly compiled code which must account for `x` being either an integer or `null`. The original deoptimization is slow, but happens once and only at the end of the function's lifetime. The end result is that while the newly optimized function avoids the deoptimization, it is in aggregate slower than the original optimized function plus the deoptimization.

For this reason, our offline profiler ignores the type information from the last type-based deoptimization in the last optimized execution of the function being profiled (in Figure 3, this would be the last type-based deoptimization in the figure). We assume that this deoptimization is from the coding idiom described above, and therefore we allow that last type-based deoptimization to remain.

Other deoptimization patterns can also occur due to very rare unexpected types or shape modifications during the execution of the application. Our technique does not consider this as a special case and records the unexpected type in the log. When this type is used in the client side to optimize the hot function, the optimizing compile might generate sub-optimal code. Therefore, there is a tradeoff between cost of executing sub-optimal code without deoptimization using offline profile information versus the collective cost of deoptimization, the cost of profiling after deoptimization, and the cost of executing sub-optimal code after the function is regarded hot again. In most cases we found that ignoring the last deoptimization was sufficient for better performance.

Reason 3. Some functions are inherently type unstable and will consistently be deoptimized no matter how much profile information is saved (often because of other kinds of deoptimizations rather than type- and shape-based deoptimizations). Optimizing these functions will result in a net loss in performance because of the constant deoptimization. We set a threshold value for number of deoptimizations (of all kinds, not just type- and shape-based) and mark all functions that exceed this threshold as non-optimizable in the program's profile annotations.

4.1.2 Stability Testing Phase

The amount of annotations added to a program as comments by the offline profiler increases the size of the program, and hence provides a cost in terms of network bandwidth when sending the program from the server to the client. We would like to minimize that cost as much as possible. The purpose of the stability testing phase is to figure out which program annotations may have unnecessary information, which we can then drop to minimize the annotation size. This phase is solely to optimize the size of the program annotations, it does not affect the client-side optimizations (either positively or negatively).

Recall that the threshold for making a function hot and sending it through the optimizing compiler is significantly higher than it might be in a statically-typed language (on the order of 1,000 invocations) in order to make it more likely that the baseline profiler has seen all of the relevant type information before optimization, thus reducing the chances of deoptimization. This threshold is very conservative because the cost of those deoptimizations is so high. The idea of the stability testing phase is to detect functions that stabilize much earlier than this threshold; for those type-stable functions we can omit the type information from the profiler's annotations (while still marking them as hot). At the client we drastically reduce the amount of time the baseline profiler is run before optimizing a function that is marked hot by the offline profiler, but still leave enough time that these type-stable functions have all of the necessary type information collected by the client engine's online profiler. In other words, we are dropping the type annotations for the type-stable functions to save space, then reconstituting them on the client side via the normal online profiling. The stability testing phase identifies the type-stable functions where we can be sure that the online profiler will get all of the necessary type information even though we are reducing the amount of time it has to profile those functions.

We define a potential type-stable function as one that, in the initial phase, was marked as hot but had no deoptimizations (except perhaps a final type-based deoptimization per the coding idiom described earlier). We detect type-stable functions empirically by rerunning the same program on the same input as for the initial phase, but taking all of the potential type-stable functions and initializing their hotness counter to a high value (rather than the normal zero), but *without* using any of the type information gathered by the initial phase. Any potential type-stable function that still does not have any deoptimizations is considered type-stable and their type annotations are removed from the program.

4.2 Client-Side Optimization

When the client receives a program containing annotations from our ahead-of-time profiler (given as code comments), it strips them from the program when that program is read into the engine's memory and stores them in an object we call the

Oracle. The Oracle stores the profile information indexed by function.

When a function is first loaded, the client engine consults the Oracle to determine if it is a hot function. If so, the function's hotness counter is initialized to a high value (rather than zero) so that it will be quickly passed to the optimizing compiler. If, on the other hand, the profile information indicates that this function is too type unstable, then the client engine marks it as unoptimizable so that it will never be passed to the optimizing compiler. These two mechanisms (the hotness counter and the unoptimizable mark) are already present in all modern JavaScript engines, and thus engines using our technique need only minor modifications to take advantage of the Oracle's information.

When a function is being compiled by the optimizing compiler, the Oracle is again consulted to gather the profiled type information. The optimizing compiler already consults the online profiler to gather type constraints in order to perform type inference; it is a simple change to have it also gather type constraints from the Oracle as well. The optimizing compiler also already uses shape information from the online profiler to inline accesses to objects (i.e., to use offset information to jump directly to a property rather than using a hash table). It is again a simple modification to have the compiler consult the Oracle to determine if a particular object access triggered a shape-based deoptimization during ahead-of-time profiling, and if so to avoid inlining the access. By avoiding this optimization we eliminate the possibility of shape-based deoptimizations at this point; while the lack of optimization slows down the code, the benefit of avoiding deoptimization provides a net gain in performance.

Consider Figure 4. This figure represents an example timeline on the client side for the same function as Figure 3 (which represented the function's execution on the server-side during profiling). We see that there is still a warmup phase, but that the function is optimized much earlier than before. The first two deoptimizations are removed, but at the expense of an unoptimized object property access to eliminate the shape-based deoptimization. Finally, the last type-based deoptimization still remains to account for the common JavaScript coding idiom described previously.

5. Evaluation

To evaluate the benefits of our ahead-of-time profiling technique, we implement it using Mozilla's production-quality JavaScript engine SpiderMonkey and test it on three different benchmark suites:

- **Octane**, the industry-standard JavaScript performance benchmark suite [7].
- **Physics**, a set of open-source JavaScript physics engines for web games [4, 8, 9, 11].
- **Membench50**, a benchmark suite consisting of real-world websites that heavily use JavaScript [5].

Note that throughout this section, "deoptimizations" refers specifically to type- and shape-based deoptimizations.

We run our experiments on an 8-core Intel i7-4790 machine with 32GB RAM running Fedora 20 Heisenbug as the profiling server and another machine with the same configuration as the client. Below, we first give details on the modifications we made to the SpiderMonkey engine for our implementation and then describe the results for each of the three benchmark suites in turn.

5.1 SpiderMonkey Modifications

We modify SpiderMonkey version 224982 from the mozilla-central repository [6] as the profiling engine and the client engine. The modified engines are available as an anonymized download. [1] We use an unmodified SpiderMonkey of the same version as the baseline to compare against. The specific modifications and heuristics that we use for the server and the client are as follows.

Server. The profiling engine is modified to log the following three classes of information along with the location in the source where they are observed. The location is defined by *{file name, line number, column number}* in the source code.

- All type-based and shape-based deoptimizations that occur during the execution of the program.
- Any newly observed types that trigger type-based deoptimizations.
- Hot method and loop compilations that are performed by the IonMonkey optimizing compiler.

We use the SpiderMonkey default of 10 iterations as the warmness threshold and 1,000 iterations as the hotness threshold. Any function that is deoptimized more than 10 times is considered a type-unstable function.

For our prototype implementation we send the profile information as a separate text file which is read by the client, rather than embedding it in the original JavaScript program. Embedding the annotations and stripping them out is trivial, but keeping them separate is more convenient for running experiments.

Client. The client is modified to read the program annotations into an oracle object. The oracle is consulted when the *JSScript* object is first created in the client engine. The warmup counter for a hot function is initialized to 950 instead of 0; thus hot functions will be compiled after 50 executions of the baseline compiled code rather than 1000.

5.2 Ahead-of-Time Profiling Cost

There is a small cost on the server-side to do the ahead-of-time profiling, though this cost is negligible. The profiler must run the program twice (once for each profiling phase) and write out the online profiler's information to disk. That

[1] https://dl.dropboxusercontent.com/u/206469/dls15.zip

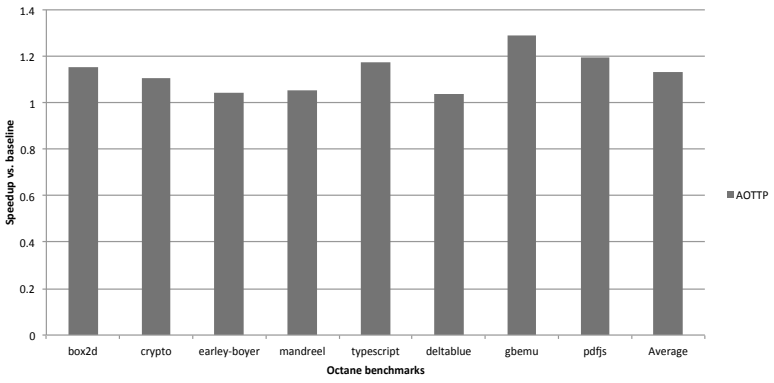


Figure 5: Average speedup of Ahead-Of-Time Type Profiling (AOTTP) versus the baseline. Higher is better.

Table 1: Number of hot functions, the total number of deoptimizations observed during the baseline and AOTTP approach, and the percent reduction in number of type and shape-based deoptimizations.

Benchmarks	# hot funcs	Baseline deopts	AOTTP deopts	% reduction in deopts
box2d	254	15	5	66.66
crypto	55	12	4	66.66
earley-boyer	69	5	2	60
mandreel	71	0	0	–
typescript	324	73	49	32.87
deltablue	66	1	1	0
gbemu	171	20	17	15
pdfjs	93	25	15	40
Average	**137.88**	**18.85**	**11.62**	**40.17**

Table 2: Program annotation size, the benchmark size without annotations, and the percent size overhead when adding the annotations to the program

Benchmarks	Profile size (kB)	Benchmark size (kB)	% overhead in size
box2d	33.9	374.01	9.08
crypto	6.57	63.91	9.97
earley-boyer	6.04	211.10	2.81
mandreel	5.61	5016.26	0.11
typescript	51.99	1254.68	4.11
deltablue	5.14	41.58	12.37
gbemu	20.22	531.99	3.80
pdfjs	16.08	1482.06	1.06
Average	**18.08**	**1121.9**	**5.41**

information is then run through the profile analyzer to parse and collate the provided information in order to create the program annotations. This analysis process takes from a few milliseconds to a few seconds over all of our benchmarks.

5.3 Octane Benchmark Suite

The Octane benchmark suite is the industry standard benchmark suite used to measure the performance of JavaScript engines. Because our technique applies only to JIT compiled code, we consider a subset of Octane benchmarks which run for a reasonable amount of time, have a significant amount of hot functions (a minimum of 50), and have deoptimizations. For the other benchmarks in the suite, ahead-of-time profiling can be avoided altogether. Benchmarks like splay and regexp focus on different parts of the engine like the garbage collector and the regular expression engine; the zlib benchmark is an `asm.js` benchmark testing the efficiency of a different compiler in the JavaScript engine; and the code-load benchmark does not exercise the optimizing compiler. Therefore, we do not consider those benchmarks. Choosing a subset

of benchmarks is justified for our approach because unlike other online compiler optimizations that are always "on", the offline profile information based optimization is optional and can be disabled for applications which do not show additional speedup.

Calculating Speedup. Octane benchmarks provide scores upon completion of individual benchmarks. The higher the score the better the performance. To calculate the speedup, we run each benchmark 22 times in different VM instances and compute the average score of the last 20 times.

Training Inputs. Octane benchmarks typically do not take in any user input. Only a few of them take specific inputs from the external world, e.g., pdf.js and typescript. For example, the typescript benchmark is a typescript compiler that compiles itself. We used the `jquery.ts` file (with minor modifications) from the TypeScriptSamples Github repository[2] as the training input instead of the typescript benchmark's regular input. For the rest of the benchmarks, we modify them with different parameters to generate the training inputs for our server. For example, the crypto benchmark was modified to encrypt and decrypt different strings and the box2d benchmark was initialized with different step parameters. In this way we ensured that the ahead-of-time profiling was always done on different inputs than the client-side performance evaluation.

Observations. Figure 5 shows the speedups obtained by our ahead-of-time type profiles (AOTTP) against the baseline implementation. The most significant improvement in the performance is seen for the gbemu benchmark where the AOTTP approach is 29% faster compared to the baseline. On average the AOTTP approach shows 13.1% improvement over the baseline configuration. Given the fact that SpiderMonkey is already highly optimized, this speedup is considered a significant performance improvement by the SpiderMonkey development team.[3]

Table 1 shows the reduction in deoptimizations when our AOTTP approach is used versus the baseline configuration. Except for mandreel and deltablue benchmarks, our AOTTP approach ensures that a significant amount of deoptimizations are avoided. On average 40.17% of the deoptimizations are eliminated using the AOTTP approach. The mandreel benchmark is generated using the Mandreel C++ to JavaScript compiler. Therefore, mandreel does not show any kind of deoptimization during the execution and is type-stable throughout the execution.

Comparing the percentage of deoptimizations reduced shown in Table 1 with the speedups numbers from Figure 5, one distinct observation would be that higher percentage of reduction in deoptimizations does not always correspond to improvement in performance. This is because, not all deoptimizations are in the critical path of execution of the benchmark. Avoiding deoptimizations that occur in a function that makes up most of the execution time would help improve performance of the application better.

The space overhead for the program annotations is minimal. Table 2 shows the size of the type profiles compared against the size of the benchmarks. On an average, the profile size is only around 5.41% of the size of the benchmarks. The typescript benchmark produces the largest profile information among all of the octane benchmarks. It produces a 51.59kB annotation which is around 4.11% of the size of the benchmark. A major chunk of the space overhead is due to the program location information which is *{file name, line number, column number}*. It is possible to drastically reduce this overhead by annotating the profile information directly in the source code instead of having a separate file.

5.4 JavaScript Physics Engines

There is no definitive way of measuring JavaScript performance when embedded in a browser, so we take two different approaches in this subsection and the next. Here we want to measure computation-heavy JavaScript code performance in a browser setting. We use four open-source JavaScript physics engine demos as our benchmarks and use frames-per-second (FPS) as our metric for evaluating performance. Our hypothesis is that ahead-of-time profiles will show an improvement in the FPS values earlier in the execution of the benchmarks, because our ahead-of-time approach allows the client to optimize hot functions much earlier during execution. We believe that these demo applications best capture the behavior of computation-heavy applications where the user expects good performance from the application from the time the application is launched.

Evaluation Setup. We evaluate 4 JavaScript physics engines: three.js [11], pixi.js [9], matter.js [4], and physics.js [8]. We use 2 demo applications from three.js and one each from pixi.js, matter.js, and physics.js, yielding a total of 5 physics engine benchmarks. These different engines have different ways of calculating FPS values, and so the x-axes of the graphs showing FPS results in Figure 6 are different. The three.js and pixi.js applications emit FPS values for every frame that is generated. Therefore, the x axis in Figures 6a, 6b, and 6c represent execution times in terms of frames displayed. The matter.js and physics.js applications emit FPS values every second. Therefore, in Figures 6d and 6e the x axis represent execution time in terms of seconds.

The benchmarks are inherently random and generate random collisions, patterns, and movements of objects for every invocation of the program. Therefore, our training input is guaranteed to be different compared to the evaluation input.

Observations Table 3 shows the improvement in FPS values for the physics engine demos during the first few seconds of execution. On average, we see 6.75% improvement in the FPS values across all the applications with three.js: canvas

[2] https://github.com/Microsoft/TypeScriptSamples

[3] Personal communication at #jsapi IRC channel

Table 3: Reduction in deoptimizations for JavaScript physics engine demo applications and the improvement in FPS values when using ahead-of-time profiling.

Benchmark	# hot funcs	Baseline deopts	AOTTP deopts	Reduction in deopts	FPS improvement
three.js:canvas ascii	99	4	3	25%	7.3%
three.js:canvas camera orthographic	71	1	1	0%	7.5%
pixi.js:3D balls	31	10	3	70%	5.9%
matter.js:multi-body collision	50	7	0	100%	6.5%
physics.js:multi-body-collision	75	33	30	9.09%	6.5%
Average	**65.2**	**11**	**7.4**	**40.82%**	**6.75%**

Table 4: Profile annotation overhead for the JavaScript physics engine demo applications.

Benchmark	Profile size (kB)	Benchmark size (kB)	Size overhead
three.js:canvas ascii	12.00	864.47	1.40%
three.js:canvas camera orthographic	8.33	846.32	1.00%
pixi.js:3D balls	3.86	225.75	1.50%
matter.js:multi-body collision	7.92	602	1.30%
physics.js:multi-body-collision	13.13	556	2.36%
Average	**9.05**	**618.9**	**1.52%**

ascii and matter.js: multi-body collision benchmarks showing significant improvement in the FPS values during the first 20–30 seconds of execution.

Figure 6 shows the FPS values seen while executing the physics engine demo applications for the first 60 seconds. The figure shows the FPS values computed for a single representative run. For example, Figure 6a shows the evolution of FPS values for the application three.js: canvas ascii. For the first 600 frames the AOTTP configuration shows better FPS values versus the baseline, then gradually the baseline FPS converges to be the same as the AOTTP. This early performance lead by AOTTP shows the effect of optimizing hot functions early. The other benchmarks have similar behavior. This is most easily seen in Figure 6d; the other benchmarks also converge, but only after several minutes. To keep the graphs legible, we only show the first 60 seconds and so for the other benchmarks the convergence point is not shown.

In some cases the FPS values for the AOTTP optimized version drops below the baseline. This drop is not due to anything inherent in the AOTTP approach, but rather is due to the inherent randomness and dynamism in the benchmarks, such as when exactly garbage collection is triggered. We ran the experiments for each of the benchmarks multiple times and observed similar average speedup in the FPS values for the AOTTP configuration.

Table 3 shows the percentage reduction in the deoptimizations for the benchmarks. On average the AOTTP approach avoids 40.82% of the deoptimizations across all the benchmarks. The benchmark pixi.js:3D balls is type stable for most parts and does not have any deoptimizations that can be avoided by AOTTP. But just identifying type-stable hot functions and compiling them eagerly yields a speedup.

Reduction in deoptimizations and the improvement in FPS values do not correlate because of multiple reasons. Some deoptimizations are more critical than others and cause major slowdowns in execution of the program. Avoiding such deoptimizations show significant improvement in performance. Also, in case of three.js:canvas camera orthograhic benchmark, there is no reduction in deoptimizations using our technique. But we see improvement in performance by simply optimizing the hot functions early based on the offline profile information.

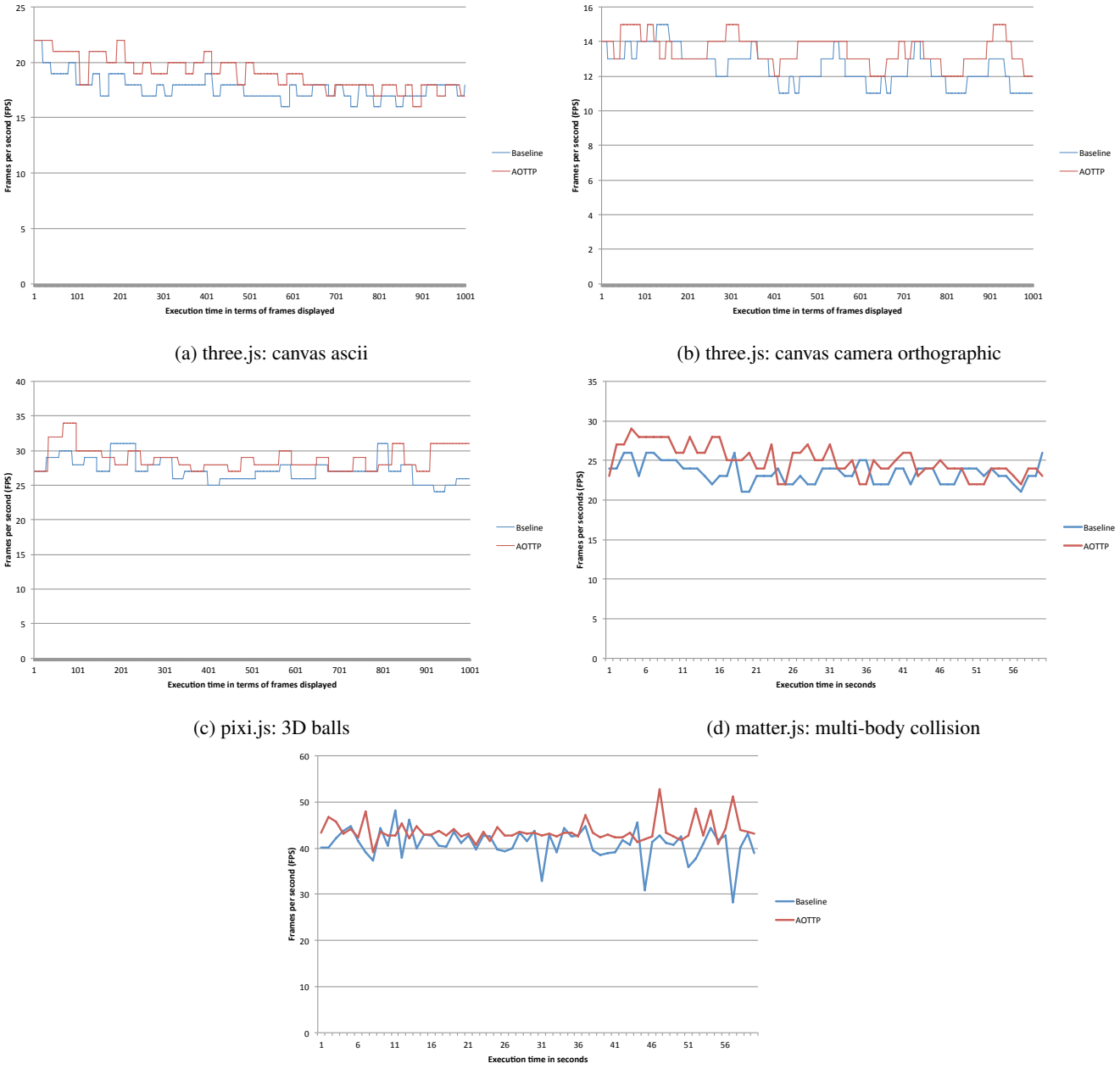
As shown in the Table 4, the space overhead for the profiles is negligible for all of the benchmarks. The average overhead is only 1.52%.

5.5 Membench50

Membench50 is a benchmark suite consisting of 50 real-world popular JavaScript-heavy web pages, primarily designed to evaluate the memory usage of the JavaScript engine. Since our optimization applies to programs that exercise the optimizing compiler, we filter out the websites that have fewer than 30 hot functions.

We use a popular automated website testing tool, Selenium IDE [10] , to simulate user interactions for these websites. For each of the websites, user interactions such as mouse clicks, key presses, and scrolling are recorded using the IDE. These interactions are saved as individual test cases and used as inputs for capturing the offline profile information. Different test cases are used to simulate the user interaction at the client side.

Performing an ideal performance analysis of our approach is difficult in this setting, because we would need to isolate the effects of our optimizations in the presence of network and IO latency in a web browser. Therefore, we present the percentage of deoptimizations that are avoided by our



(a) three.js: canvas ascii

(b) three.js: canvas camera orthographic

(c) pixi.js: 3D balls

(d) matter.js: multi-body collision

(e) physics.js: multi-body collision

Figure 6: Frames per second (FPS) plots for 5 JavaScript physics engine benchmarks. The x axis for Figures 6a, 6b, and 6c represent execution times in terms of frames displayed. The x axis of Figures 6d and 6e represents execution time in seconds. Higher is better.

approach as a metric to indicate the effectiveness of our optimization.

Table 5 shows the percentage reduction in deoptimizations using the AOTTP approach. In general, most of the benchmarks show reduction in the deoptimization counts while using the AOTTP technique. On average there is a 33.04% reduction in deoptimizations across all benchmarks, with an average profile size of less than 13.08kB. The size of the JavaScript code in these websites are in the order of MB. Therefore, the overhead of profile size is negligible compared to the size of the website. To give a rough estimate, all of the websites present in this benchmark suite use the jquery library. The average profile size is 13% of the size of the jquery library. Therefore, it is safe to assume that the space overhead

Table 5: Experimental results for a subset of Membench50 benchmark suite: number of hot functions, number of deoptimizations in the baseline, number of deoptimizations using AOTTP, percentage reduction in the deoptimizations, and size of the aggregate profile collected using AOTTP approach.

Benchmarks	# of hot funcs	# Baseline deopts	# AOTTP deopts	Reduction in deopts	Profile size (kB)
businessinsider.com	393	44	20	54.54%	84.05
lufthansa.com	30	12	10	16.66%	3.81
amazon.com	104	23	13	43.47%	21.18
tbpl.mozilla.org	30	17	11	35.29%	2.12
taobao.com	49	18	16	11.11%	4.62
nytimes.com	87	27	24	11.11%	2.27
cnn.com	208	22	18	18.18%	19.76
bild.de	42	2	1	50%	8.49
spiegel.de	32	13	10	23.07%	2.40
lenovo.com	34	0	0	–	3.17
weibo.com	44	4	1	75%	2.37
ask.com	40	4	3	25%	2.82
Average	**91.08**	**15.5**	**10.58**	**33.04%**	**13.08**

Table 6: Percentage of type- and shape-based deoptimizations (TSDeopt) and percentage of unclassified deoptimizations (UDeopt) verses total number of deoptimizations for Octane benchmark suite.

Benchmarks	All Deopts	TSDeopt	UDeopt
box2d	40	40%	27.5%
crypto	24	33.33%	20.8%
earley-boyer	42	9.5%	9.5%
mandreel	3	0%	100%
typescript	595	9.7%	25.8%
deltablue	4	25%	75%
gbemu	112	17%	11.6%
pdfjs	73	17.8%	17.8%
Average	**111.62**	**19.04%**	**36%**

of the ahead-to-time profile for all of these benchmarks is negligible.

5.6 Kinds of Deoptimizations

Our technique focuses on type- and shape-based deoptimizations, though other kinds of deoptimizations can happen. We rely on the JavaScript engine to classify each deoptimization for us, in order to determine which ones to handle with our profiling technique. One difficulty we encountered specific to SpiderMonkey is that, for technical reasons having to do with the engine's implementation, there are some deoptimizations that the engine leaves unclassified. In other words, for these deoptimizations we do not know whether they were type- or shape-based or some other kind of deoptimization. Our implementation conservatively assumes that they are not type- or shape-based and ignores them. Table 6 shows for each Octane benchmark during ahead-of-time profiling the total number of deoptimizations, the percentage that were classified as type- or shape-based, and the percentage that were left unclassified.

We see that a significant portion of the deoptimizations are classified as type- or shape-based, but that an even larger portion are left unclassified. Based on our experience we conjecture that many of these unclassified deoptimizations are actually type- or shape-based and would be amenable to our technique if we could recognize them. However, doing so would require much more extensive changes to the profiling engine (but *not* the client engine). Considering that we get a significant performance benefit just from handling the identified type- and shape-based deoptimizations, it is likely that extending our technique in this way would yield even more significant performance gains.

6. Conclusion

We have described a technique to optimize JavaScript programs sent from a server to a client by performing ahead-of-time profiling on the server side in order to reduce deoptimizations on the client side. We have shown that these deoptimizations are an important concern for performance, and that reducing the deoptimizations provides a significant performance benefit. Besides reducing deoptimizations, our technique also allows hot functions to be compiled much earlier than they normally would *and* without having to fear increased deoptimizations due to the reduced profiling time entailed by early compilation.

Our technique relies on several key insights to be practical and effective, such as identifying the correct information to profile to tradeoff the costs and benefits of type profiling and identifying common coding idioms that directly impact the effectiveness of profiling.

We evaluate our technique over three sets of benchmarks: the industry-standard Octane benchmark suite, a set of JavaScript physics engines, and a subset of real-world websites from the Membench50 benchmark suite. Our evaluation shows a maximum speedup of 29% and an average speedup

of 13.1% for Octane benchmarks, a maximum improvement of 7.5% and an average improvement of 6.75% in the FPS values for JavaScript physics engines, and an average 33.04% reduction in deoptimizations for the Membench50 benchmarks. The collected profile information is on an average 4% of the size of the JavaScript code.

Acknowledgments

We would like to thank the the anonymous reviewers and developers at Mozilla for their valuable input. The work described in this paper was significantly supported by Qualcomm Research.

References

[1] Spidermonkey baseline compiler. `https://wiki.mozilla.org/Javascript:SpiderMonkey:BaselineCompiler`, 2015.

[2] Common language runtime. `http://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx`, 2015.

[3] Java virtual machine. `http://docs.oracle.com/javase/specs/jvms/se7/html/`, 2015.

[4] Matter.js - a 2d rigid body javascript physics engine. `http://brm.io/matter-js/`, 2015.

[5] Membench50. `http://gregor-wagner.com/tmp/mem50`, 2015.

[6] Mozilla central repository. `https://hg.mozilla.org/mozilla-central`, 2015.

[7] Octane benchmark suite. `https://developers.google.com/octane/`, 2015.

[8] Physics.js physics engine. `http://wellcaffeinated.net/PhysicsJS/`, 2015.

[9] pixi.js 3d rendering engine. `http://www.pixijs.com`, 2015.

[10] Selenium ide. `http://docs.seleniumhq.org/docs/02_selenium_ide.jsp`, 2015.

[11] three.js physics engine. `http://threejs.org`, 2015.

[12] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[13] M. Arnold, A. Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.

[14] asm.js specification. asm.js specification. `http://asmjs.org/spec/latest/`, 2015.

[15] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 2009.

[16] Closure Compiler. Google Inc. closure compiler. `https://developers.google.com/closure/compiler/`, 2015.

[17] Crankshaft compiler. V8 engine. `http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler`, 2015.

[18] Flow. Flow static type checker. `http://flowtype.org`, 2015.

[19] L. Guckert, M. OConnor, S. Kumar Ravindranath, Z. Zhao, and V. Janapa Reddi. A case for persistent caching of compiled javascript code in mobile web browsers. In *In Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2013.

[20] U. Hölzle and D. Ungar. A third-generation self implementation: reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, 1994.

[21] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, 1991.

[22] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, 1992.

[23] JavaScriptCore. Webkit JavaScriptCore virtual machine. `http://trac.webkit.org/wiki/JavaScriptCore`, 2015.

[24] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2003.

[25] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[26] J. Oh and S.-M. Moon. Snapshot-based loading-time acceleration for web applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[27] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, 2001.

[28] SpiderMonkey. SpiderMonkey JavaScript Engine. `http://www.mozilla.org/js/spidermonkey/`, 2015.

[29] TypeScript. Typescript. `http://www.typescriptlang.org`, 2015.

[30] V8. Google Inc. V8 JavaScript virtual machine. `https://code.google.com/p/v8`, 2015.