Workload Characterization of Server-Side JavaScript

Takeshi Ogasawara
IBM Research – Tokyo
Tokyo, Japan
Email: takeshi@jp.ibm.com

Abstract—**Recently, scripting languages are becoming popular as languages to develop server-side applications. Modern JavaScript compilers significantly optimize JavaScript code, but their main targets are client-side Web applications. In this paper, we characterize the runtime behaviors of server workloads on an emerging JavaScript server-side framework, Node.js, comparing it to client-side JavaScript code. The runtime profile shows that a large amount (47.5% on average) of the total CPU time is spent in the V8 C++ library used for server workloads, while only 3.2% is for client-side programs. Such high CPU usage in the C++ code limits the performance improvements for server workloads, since recent changes in the performance of client-side workloads are due to optimized JavaScript code. Our analysis of complex calling contexts reveals that function calls to the V8 runtime from the server-side framework to handle JavaScript objects significantly contribute to the high CPU usage in the V8 library (up to 22.5% of the total time and 15.4% on average). We also show that the function calls to the V8 runtime from compiled JavaScript code are another source of the high CPU usage in the V8 library code (up to 24.5% of the total and 18.7% on average). Only a few JavaScript functions that implement the server-side framework API make these function calls to the V8 runtime and contribute to a large amount of the CPU time (up to 6.8% of the total).**

I. Introduction

Scripting languages such as JavaScript [1] and Python [2] rely on dynamic typing. A single operation can handle multiple data types since the variables do not have types when the programmers write the code and might point to objects of any type. When the code is executed, the type of the target data is dynamically resolved and an appropriate operation is selected for the type. For example, if a program accesses a property x of the data, then how to retrieve x from the data depends on the type of the data and can only be determined at runtime. In contrast, with static typing (as in C), the type is fixed for each variable when the code is written. The specialized code is written for different types of data.

Dynamic typing simplifies the programming while adding runtime overhead for resolving the types, which is not required with static typing. Suppose that the code accesses a property x of some data in a scripting language. The language system resolves the type of the data, resolves an access method that can locates x for the resolved type, and executes it. The runtime overhead of resolving the types is the cost of the resolution processes.

Scripting languages are becoming increasingly popular for developing both client-side applications and server-side applications. One powerful scripting languages is JavaScript, which is a widely used language to used to develop Web applications. To browse webpages more efficiently, modern compiler optimization is continuously evolving ways to reduce the overhead of dynamic typing. However, these compilers are typically targeting the client-side code. Our research interests involve characterizing the runtime behaviors of server-side JavaScript workloads to evaluate the opportunities of compiler optimization that is effective for the client-side code in the server-side workloads.

In this paper, we characterize the runtime behaviors of some server-side JavaScript programs. We use the V8 JavaScript engine [3] as our target compiler and Node.js [4] as our target server-side framework. Through comparisons to the runtime behaviors of client-side JavaScript programs, we assess the differences in server-side workloads versus client-side workloads so the runtime behaviors can be optimized by the JavaScript engine.

In summary, the contributions of our paper are as follows:

- Comparisons of the CPU times that a JavaScript compiler can optimize between server-side workloads and client-side workloads using the Node.js framework. Our analysis showed that the targets of compiler optimization are limited to only 20.3% of the total execution time for server workloads in the worst case (32.5% on average). This ratio is far less than that of the client-side code (83% on average).
- Analysis of the dynamic behaviors of the server-side JavaScript code that reduce the scope of compiler optimization. A large amount of the CPU time is spent in the V8 library code written in C++ (up to 52.4% of the total time, 47.5% on average). The V8 compiler optimizes JavaScript code based on the runtime feedback but does not optimize the C++ code call trees. More than 1,600 functions of the V8 library are executed in the server-side workloads and most of the functions consume less than 1% of the total CPU time. The JavaScript compiler does not address the overheads of many function calls and does not specialize the function calls by using the runtime type feedback.
- Identification of the characteristics of server-side workloads and frameworks that cause excessive CPU usage in the V8 library code. We showed that the excessive CPU usage is due to function calls of the V8 runtime, not only from the compiled JavaScript code but also from the C++ code of a server-side JavaScript framework to handle JavaScript objects. The cost of executing the V8 runtime from the C++ server framework code, which the current JavaScript compilers cannot optimize, appear to be a bottleneck in the performance of server workloads.

- Analysis of the cost of function calls to the V8 runtime from compiled JavaScript code. When we ranked the V8 runtime functions in terms of CPU consumption, a top function consumed up to 6.8% of the total CPU time (4.2% on average). The function is called from the JavaScript code of the Node.js API.

The rest of the paper is organized as follows. Section II explains the runtime overheads of the scripting languages as mainly caused by the dynamic typing and explains how compilers can reduce it to improve the overall performance. Section III presents our detailed analysis of the runtime behavior of server-side JavaScript compared to that of the client-side code. We emphasize the runtime behaviors that are observed only in the server-side code. This allows us to identify the factors slowing down the server-side code. We discuss related work in Section IV. We conclude the paper in Section V.

II. The runtime overhead of scripting languages

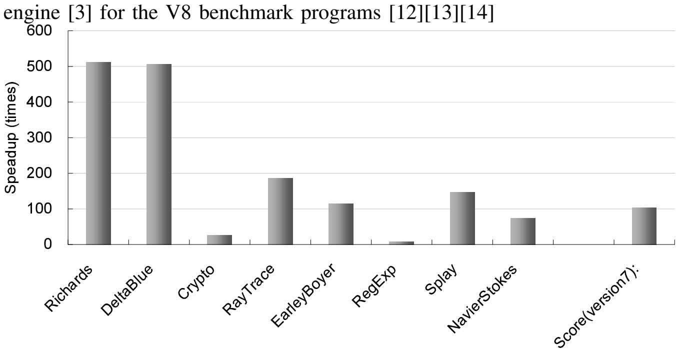
Scripting languages such as JavaScript and Python use dynamic typing extensively for higher programmer productivity. With dynamic typing, the types of the objects accessed at a given program location are resolved at runtime. Programmers write the code without declaring the types of the variables. Since the variables are type-free, the types of the objects pointed by the variables cannot be determined in the source code. The object types are checked whenever the objects are accessed at runtime.

Dynamic typing adds significant penalties to the runtime performance of scripting languages [5]. In general, if the types are determined for the objects, how to access the properties of objects can be easily determined. For example, an expression o.x in a program retrieves the property with name x from an object pointed to by a variable o. In dynamic typing, this retrieval is performed at runtime since the types are dynamically determined. When this expression is executed, the runtime code of the scripting languages (an interpreter of the byte code or the compiled code itself) must obtain the type of the object currently pointed to by o and perform the retrieval during the program execution. In addition, the cost of the retrieval is very high and is a major reason that scripting languages run slower than statically-typed languages. The runtime code has to determine what retrieval method to use for the current type (e.g., searching a hash map or calling a getter/setter function) and then executes it whenever the expression is executed during the program execution.

In static typing, there are no such runtime costs, since the retrieval method can be determined at compile-time. For an expression o.x in a program in a statically typed language such as Java, the runtime code simply accesses the data in the current object. This simple access is possible because the data structure of the object is fixed for each access. The variable o has a type, and therefore the objects pointed to by o have that type. Once a type (called a class in Java) is given, the retrieval method is fixed (with a constant offset in the object for x by field resolution), and the calculation is only done once before the program begins execution (where the offset is based on the memory layout of the class) [6].

Inline caching [7] is a software technique that caches the result of the dynamic retrieval of the symbolic reference for each program location (e.g., o.x). With inline caching, there is a separate cache that is associated with each program location that makes a symbolic reference. When the runtime code needs a symbolic reference at a program location, it checks if the associated cache has a previous result from a dynamic retrieval. If the cache has a previous result, then the runtime code reuses it so that the retrieval is skipped. To test for such cache hits, the cache has a pair of an object type that was previously tested (the type of an object pointed to by o) and the retrieved access method for that reference. In an efficient implementation, the code that tests cache hits and accesses to the actual data will be a small number of machine instructions. Similar inline caching techniques have been proposed for optimizing other dynamic retrieval systems such as type inclusion testing [8][9].

Fig. 1. Performance improvement by inline caching on the V8 JavaScript engine [3] for the V8 benchmark programs [12][13][14]



The efficiency of inline caching relies on an empirical observation that the same code tends to test the same object type [7]. As long as the types of the tested objects are the same as the cached type, the runtime code can complete the symbolic reference with just a cache lookup. Inline caching can be extended to cache more than one result per program location when multiple types are frequently tested [10].

Inline caching can greatly improve the performance of scripting-language applications [11]. For example, we observed about a 100-fold speedup (geometric mean, with a max around 500-fold) in JavaScript benchmark programs, based on the scores with and without inline caching [1] (Fig. 1).

III. Performance analysis of JavaScript workloads

In this section, we first describe the JavaScript engines that have been evolving for Web browsers and then the emerging applications of JavaScript on the server side. Most of the current JavaScript compilers focus primarily on improving the execution speed of client-side JavaScript. Then we show that the behaviors of typical server-side applications are quite different from the client-side programs.

To measure the performance of JavaScript code, we used Node.js (version 0.11.9) with a V8 JavaScript engine (version 3.22.24.5). We used the same V8 engine for clients-side

[1]The command-line option nouse_ic was used with the V8 JavaScript engine to measure the scores without inline caching. We used V8 version 3.22.24.5 for this measurement, which is a V8 version that is used by the latest version of Node.js (version 0.11.9) and which runs faster than the latest V8 version available for our environment at the time of writing.

workloads and server-side workloads. Our test machine used the Fedora 20 Linux operating system on an 3.50-GHz Intel Xeon E3-1270v2 with 32-GB physical memory. To collect the runtime profiles of the CPU times for the programs, we used the standard Linux tool called perf, which samples the CPU cycles and function call chains.

For the analysis of the client-side benchmark, we collected the runtime profiles for the benchmark kernel code. For the analysis of the server-side code, we collected the runtime profiles for 30 seconds for the simple workloads and for 60 seconds for the complex workloads (starting after the performance became stable).

A. JavaScript Engines Optimized for Web Browsers

Scripting languages are becoming increasingly popular to develop server-side applications. In particular, JavaScript is one of the major script languages. JavaScript is ubiquitous since standard Web browsers routinely execute JavaScript code from the Internet. The performance of JavaScript engines has been addressed to improve the efficiency of Web browsing. V8 JavaScript engine (V8) [3] is one of the most efficient JavaScript engines, which is widely used in Googles Web browsers. V8 compiles all of the JavaScript code to native code when functions are first run. When V8 identifies the code as hot, it optimizes the code with another optimizing compiler by using the run-time type feedback [15].

B. Server-Side JavaScript Frameworks

JavaScript is becoming popular to write applications that run on the server side as more programmers use JavaScript to develop webpages that run on the client side. Node.js [4] is one of the most successful server-side JavaScript frameworks in the real world. Node.js is an event-based programming framework for JavaScript, which can highly scale for massive network connections. I/O operations are non-blocking in Node.js. Node.js invokes event handlers that programmers write in JavaScript when the pending I/O data becomes ready. Node.js runs JavaScript code on V8. Node.js handles network requests in a single thread to reduce the overhead of context switching and the memory footprints. Node.js applications are developed by using the built-in functions of Node.js and the external packages that are managed in the open-source community (e.g., DB connectors and Web application frameworks).

C. The Runtime Behaviors of Client-Side JavaScript Code

Dynamic compilers of JavaScript increasingly improved the performance of JavaScript. Such improved performance has enabled a variety of webpages (more complex code can achieve richer services and user experiences). The compilers can improve the performance of the programs when the compiled code runs for most of the program execution time.

For the client-side programs, most of the program execution times were spent in the compiled code, therefore, JavaScript compilers can optimize the overall performance of the programs. Fig. 2 shows the CPU time breakdown of the V8 benchmark on the V8 engine. The compiled code consumes 83.1% of the total CPU time on average.

For some of the programs, another key component is the V8 shared library (libv8.so), which is written in C++. Since this component is not the JavaScript code, the JavaScript compiler does not optimize the V8 shared library code. The CPU time spent in libv8 is 3.2% of the total CPU time on average. With further analysis of the CPU time spent in libv8, a most time consuming component of libv8 is garbage collection (GC) for Earley-Boyer and Splay [12], the library routines for regular expressions for RegExp, and the code that extends the property arrays of JavaScript objects for Raytrace. For RegExp, the regular expression routines call `memchr` and `memcpy`, which consume most of the CPU time spent in libc.

D. Examples of Server-Side JavaScript Applications

We used TechEmpower Web Framework Benchmarks [16] to analyze the server-side JavaScript workloads. The benchmark programs measure the performance of server-side software stacks including Node.js by executing fundamental tasks such as JSON serialization, database access, and server-side template composition. The database workloads use MongoDB [17] and MySQL and look up random records or update them by using the native database drivers or the object mappers.

Also we used an application that performs more complex transactions than Web Framework Benchmarks. Acme Air [18] is built on the top of Node.js. The application implements a fictitious airline to demonstrate key business requirements such as the scalability to massive API calls, deployment in clouds, and support of mobile connections. The users can log in to their accounts, search for flights, book flights, and manage the bookings. The application consists of two tiers: a Node.js part and a database part. Acme Air uses MongoDB to manage flights and user account. Also Acme Air uses external Node.js packages (e.g., express) developed in the open-source community of Node.js. We used a real-world scenario that the application provides for benchmarking with the JMeter benchmark driver [19].

We ran the database services and benchmark drivers on a separate Fedora 20 machine. The machine has two 2.9-GHz Xeon E5-2690 chips (including 32 hardware threads on 16 cores) with 96-GB physical memory. The machine and the test machine were connected using the gigabit Ethernet. The database services and drivers required a little CPU resource on the machine (only less than 5% of the total CPU resource). Also the workloads required a little network resource (only less than 2% of the total network bandwidth). The server workloads fully utilized a single hardware thread of the test machine except two workloads that used the MySQL native library.

E. The Runtime Behaviors of Server-Side JavaScript Code

To analyze how much CPU time the V8 compiler can target for server-side code, we analyzed the CPU time breakdown for the workloads by using the Linux perf tool. Fig. 3 shows the CPU time breakdown for Web Framework Benchmarks and Acme Air. The CPU times consumed in the compiled code are only 20.3-40.2% (32.5% on average). In contrast, the CPU times consumed in the V8 shared library (libv8) are more than the CPU time spent in the compiled code, 43.6-52.4% (47.5% on average). Libc consumed 2.4-5.4% (3.4% on average) of the total CPU time. Node, which is written in C++, consumed 0.25-5.5% (2.7% on average) of the total CPU time. The kernel handled the network traffic (between the Node server and

Fig. 2. CPU time breakdown per module for V8 benchmark programs

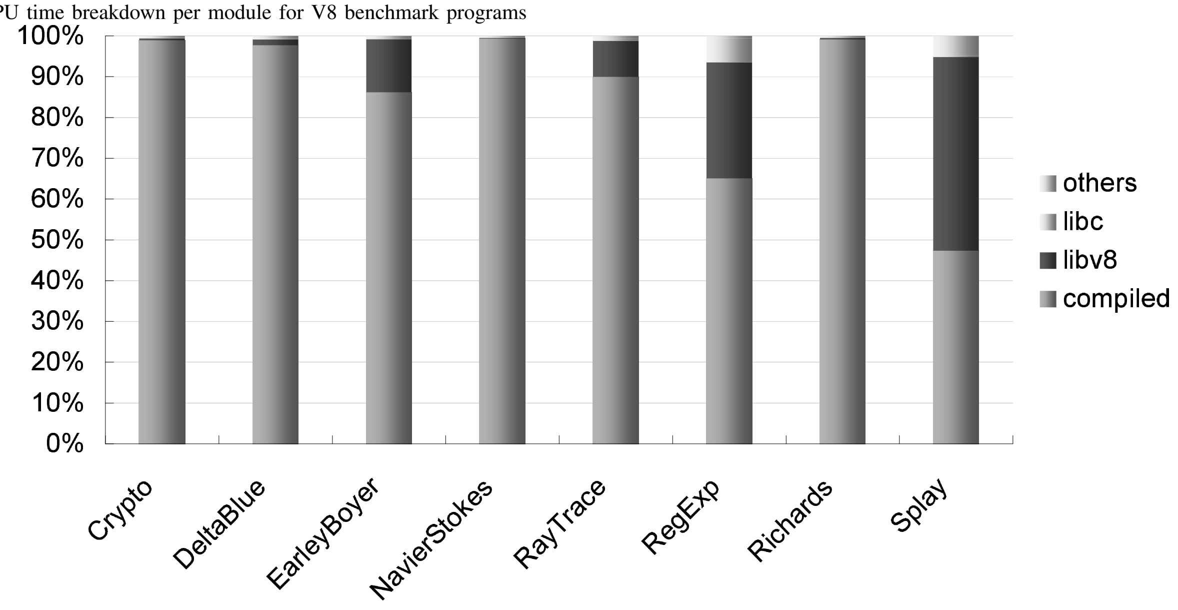
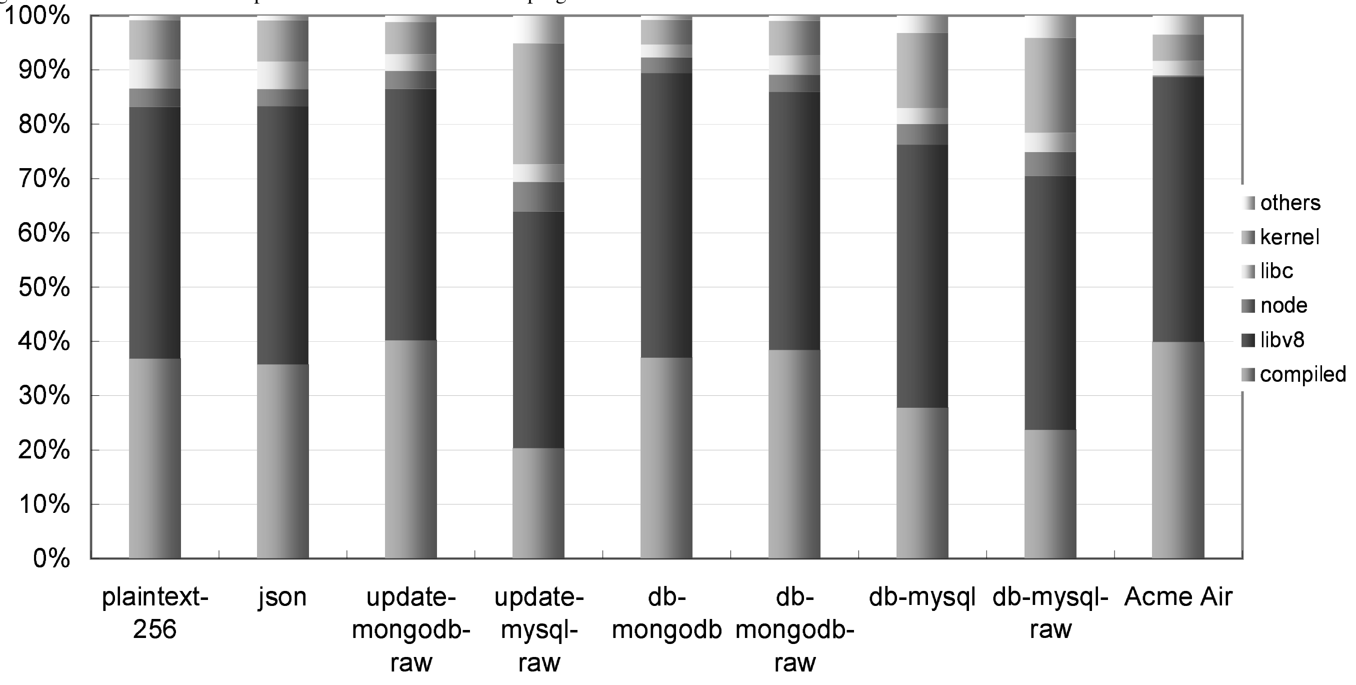


Fig. 3. CPU time breakdown per module for V8 benchmark programs



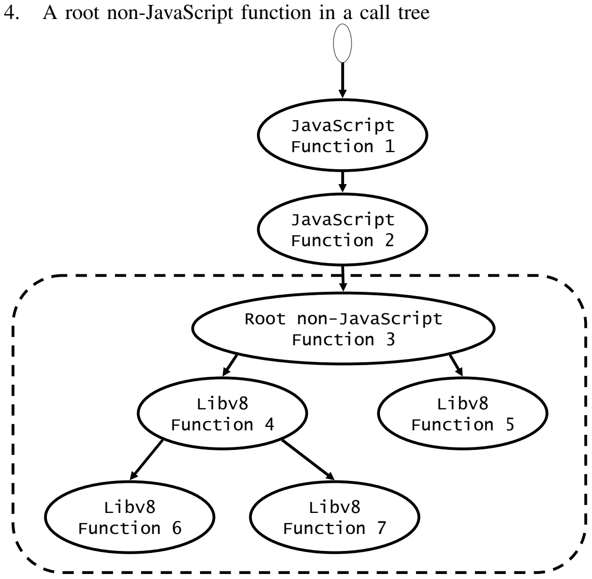
the database or between the Node server and the benchmark driver) consumed 4.6-22.3% (8.6% on average). The remaining modules (others in the figure) consumed 0.74-5.1% (1.7% on average). For the workloads using the MySQL native library (mysql-raw in the figure), the native library consumed 2.3-3.7% of the total CPU time.

The CPU time breakdown shows that the compiler can optimize only 32.5% of the program execution time for the server workloads. The ratio is smaller than 83.1% for the client-side programs discussed in Section III-C. As we discussed in Section II, JavaScript compilers improved the overall performance of the client-side programs by optimizing the JavaScript code. Therefore, we need additional JavaScript optimization techniques to improve the overall program performance of the server-side workloads.

F. Analysis of the CPU Time Spent in the V8 Shared Library

We analyzed the CPU times spent in the V8 shared library (libv8) to address the problem of the lower CPU time spent in the compiled code than the client-side programs. We ran the perf tool to collect the samples of CPU time cycles when the performance of the server workloads were stable (compilations rarely occurred). We can compare the CPU times used for the software components within each workload. However, we cannot directly compare the CPU times spent in a specific module between workloads since the throughputs of request

Fig. 4. A root non-JavaScript function in a call tree



handling were not the same across the benchmark programs. The high CPU usages in the software components can be due to high throughputs.

1) CPU Time Samples per Libv8 Function: The perf tool sampled the CPU cycles for more than 1,600 functions of libv8. Most of the functions consumed less than 1%.

2) A Methodology of CPU Time Breakdown by Identifying the Root Non-JavaScript Functions in Call Stacks: To analyze how the libv8 functions are called in the server-side workloads, we calculated the CPU times that were spent in special call trees, whose roots are *root non-JavaScript functions*. Fig. 4 shows an example of the special call trees in a dashed box. Function 3 is a root non-JavaScript function. We focus on the CPU times spent in the special call tree. We calculated the CPU time for each special call tree by using the call stack sampling feature of the perf tool. To traverse call chains correctly, we built our target binary modules such as Node.js and V8 with a compiler option `-fno-omit-frame-pointer` to ensure the call chains created in the optimized code of the target modules.

Root non-JavaScript functions are the functions that are called from the last called JavaScript code. If no JavaScript code is executed in the call stack, root non-JavaScript functions are the last called Node API functions. Fig. 5 shows an example call stack sampled by perf. The last called function is a libv8 function, LocalLookupRealNamedProperty(), and is shown at the bottom since the stack grows from the top to the bottom in the figure. The perf tool sampled 26906833 CPU cycles while the code of LocalLookupRealNamedProperty() was running. The hexadecimal numbers such as 0x2a80c4821f57 in the middle of the figure are the addresses of the compiled JavaScript code. A libv8 function, Runtime_GetProperty(), is a root non-JavaScript function since it is the function that was called from the JavaScript code at the address 0x2a80c80824e, which is the last called JavaScript code.

We calculated the CPU times of the root non-JavaScript functions as follows. For each call stack sampled by perf, we identified the root non-JavaScript function and then added the

Fig. 5. A call stack example for a server-side JavaScript workload

```
Relative Ratio          CPU Cycles
0.01%                   26906833
__libc_start_main
node::Start(int, char**)
uv_run
uv__io_poll
uv__stream_io
uv__read
node::StreamWrapCallbacks::DoRead(uv_stream_s*, long,
    uv_buf_t const*, uv_handle_type)
v8::Function::Call(v8::Handle<v8::Value>, int,
    v8::Handle<v8::Value>*)
v8::internal::Execution::Call(v8::internal::Isolate*,
    v8::internal::Handle<v8::internal::Object>,
    v8::internal::Handle<v8::internal::Object>, int,
    v8::internal::Handle<v8::internal::Object>*, bool*, bool)
0x2a80c4821f57
0x2a80c484913e
...
0x2a80c4e815a3
0x2a80c480824e
v8::internal::Runtime_GetProperty(int,
    v8::internal::Object**, v8::internal::Isolate*)
v8::internal::Runtime::GetObjectProperty(
    v8::internal::Isolate*,
    v8::internal::Handle<v8::internal::Object>,
    v8::internal::Handle<v8::internal::Object>)
v8::internal::Object::GetPropertyWithReceiver(
    v8::internal::Object*, v8::internal::Name*,
    PropertyAttributes*)
v8::internal::JSReceiver::Lookup(v8::internal::Name*,
    v8::internal::LookupResult*)
v8::internal::JSReceiver::LocalLookup(
    v8::internal::Name*,
    v8::internal::LookupResult*, bool)
v8::internal::JSObject::LocalLookupRealNamedProperty(
    v8::internal::Name*, v8::internal::LookupResult*)
```

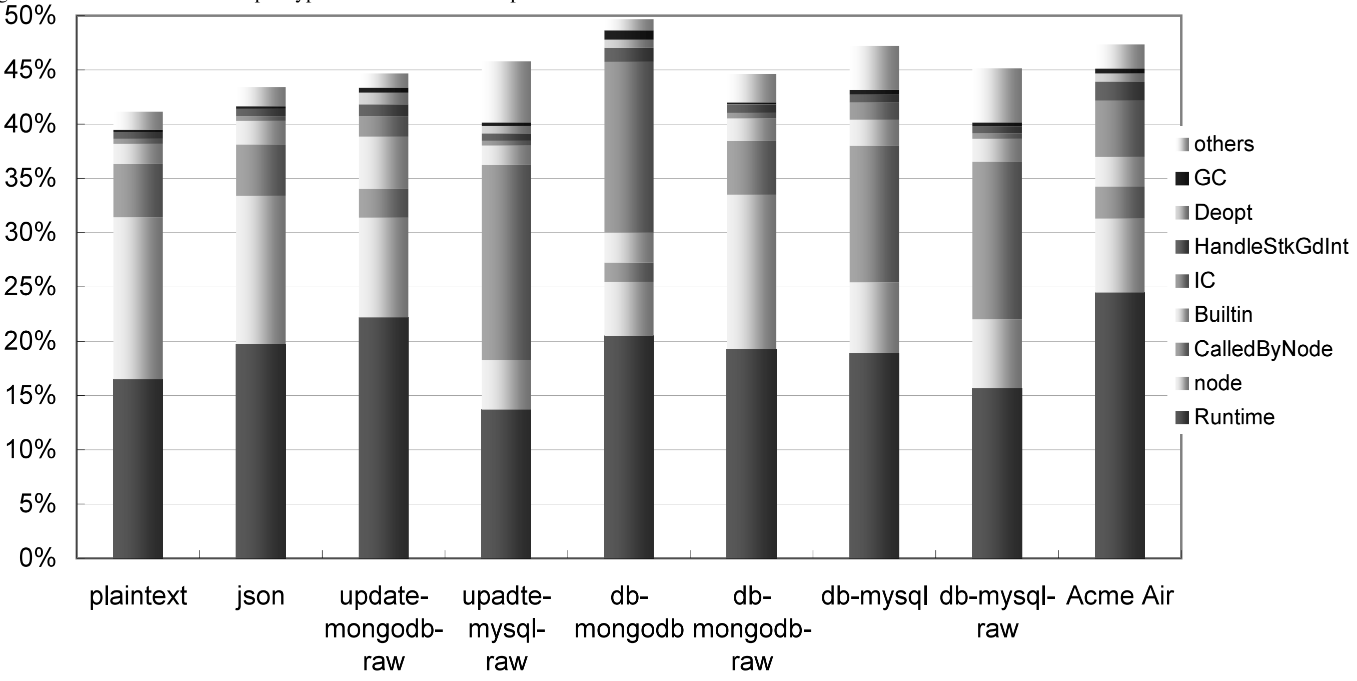
CPU cycles sampled for the call stack to the CPU time of the root non-JavaScript function. In the above example, 26906833 CPU cycles were added to the CPU time of the root non-JavaScript function, Runtime_GetProperty().

We analyzed why the excessive CPU time is spent in libv8 by analyzing the root non-JavaScript functions and their CPU times. Root non-JavaScript functions with high CPU times represent the library code that is frequently called by the JavaScript code and causes the excessive CPU times in libv8.

3) CPU Time Breakdown with Types of Root Non-JavaScript Functions: We grouped hundreds of root non-JavaScript functions into several groups based on their functionalities: the functions of the Runtime V8 class (Runtime in Fig. 6), the functions of the Built-in V8 class (Built-in), the functions of the Node.js native components (node), the functions that were called by the Node.js natives (CalledByNode), the functions of the V8 classes for inline caching (IC), garbage collection (GC), etc. Fig. 6 shows the CPU time breakdown for libv8 with the groups (the ratio to the total CPU time).

We first overview the CPU times spent in the groups and then investigate each group in the following sections. When we ranked the groups in terms of the CPU time, Runtime, Node, and CalledByNode are top three groups. The three groups consumed a majority of the CPU times spent in libv8: 54.8-88.3% of the CPU time spent in libv8 (77.8% on average). Also the groups consumed a significant part of the total CPU time: 27.2-38.5% (35.3% on average). When we focus on two groups that are unique to Node.js (Node and CalledByNode), the CPU times spent in the groups are 13.6-49.2% of the CPU time spent in libv8 (34.0% on average).

Fig. 6. CPU time breakdown per type of root non-JavaScript functions



The first group of the top three groups is Runtime. The functions of the Runtime group provide generic operations for JavaScript objects to the compiled JavaScript code. When the compilers did not optimize the function calls to the Runtime functions for the JavaScript code, the function calls increase the CPU time spent in the group. The CPU times spent in the group are a major part of the CPU time spent in libv8 (29.9-51.7% and 41.3% on average) and also affected the total CPU time (13.7-24.5% and 18.7% on average).

The second group is Node. The root non-JavaScript functions for the group are the C++ code of Node.js that implements the Node API (e.g., node::StreamWrap::WriteBuffer()). When the JavaScript code calls the Node.js API functions, the function calls increase the CPU time spent in the group. Since the compilers understand only the semantics of JavaScript, they cannot optimize the Node.js API. The CPU times spent in the Node group are another major part of the CPU time spent in libv8 (10.0-36.3% and 18.1% on average) and are not negligible in the total CPU time (4.6-15.0% and 8.2% on average).

The third group is CalledByNode. This group is similar to the Node group, but no JavaScript code is executed in the call stack. Therefore, no code in the call stack is optimized by the V8 compilers. The root non-JavaScript functions for the group are the C++ code of Node.js that implements the Node API. Fig. 7 shows an example call stack for the group. A function node::StreamWrapCallbacks::DoRead() is the root non-JavaScript function in the call stack. The CPU times spent in the group (1.8-18.0% and 5.6% on average) are not negligible in the total execution time. However, the JavaScript compilers cannot optimize them. The group consumed 3.6-39.3% of the CPU time spent in libv8 (12.3% on average).

The rest of the groups consumed a small fraction of the CPU times and will not affect the overall program performance. The CPU times spent in the GC group, which are the overheads of garbage collection, are only 0.44-1.74% of the CPU time spent in libv8 (0.76% on average) and 0.19-0.86% of the total CPU time (0.34% on average). The CPU times for the Inline Caching (IC) group, which are the costs of cache miss handling and cache maintenance, are 2.68% of the CPU time spent in libv8 and 1.22% of the total CPU time. The low CPU consumption represents the high efficiency of inline caching

Fig. 7. An example of the CalledByNode group

```
Relative Ratio        CPU Cycles
0.02%                 21546818
__libc_start_main
node::Start(int, char**)
uv_run
uv__io_poll
uv__stream_io
uv__read
node::StreamWrapCallbacks::DoRead(uv_stream_s*, long,
    uv_buf_t const*, uv_handle_type)
v8::Object::Get(v8::Handle<v8::Value>)
v8::internal::GetProperty(v8::internal::Isolate*,
    v8::internal::Handle<v8::internal::Object>,
    v8::internal::Handle<v8::internal::Object>)
v8::internal::Runtime::GetObjectProperty(
    v8::internal::Isolate*,
    v8::internal::Handle<v8::internal::Object>,
    v8::internal::Handle<v8::internal::Object>)
v8::internal::Object::GetPropertyWithReceiver(
    v8::internal::Object*, v8::internal::Name*,
    PropertyAttributes*)
v8::internal::JSReceiver::Lookup(v8::internal::Name*,
    v8::internal::LookupResult*)
v8::internal::JSReceiver::LocalLookup(
    v8::internal::Name*, v8::internal::LookupResult*,
    bool)
v8::internal::JSObject::LocalLookupRealNamedProperty(
    v8::internal::Name*, v8::internal::LookupResult*)
int v8::internal::Search<(v8::internal::SearchMode)1,
    v8::internal::DescriptorArray>(
    v8::internal::DescriptorArray*,
    v8::internal::Name*, int)
```

for most of the server-side workloads. Only db-mongodb and Acme Air workloads show the high CPU consumption for the group. For db-mongodb, the components of the IC group that handled cache misses consumed 3.1% of the total CPU time for loads and 9.6% for calls. For Acme Air, the IC group components that handled cache misses consumed 3.6% of the total CPU time for loads and 0.7% for stores. The Built-in group consumed only 1.8-4.8% of the total CPU time (2.4% on average). For the update-mongodb-raw workload, two types of the functions in the Built-in group consumed more CPU time than the other workloads: 2.3% for HandleAPICall function and 2.4% for the array operation functions.

4) Node and CalledByNode Groups: The root non-JavaScript functions for the Node and CalledByNode groups are the Node API functions. The Node API functions are the C++ code. In addition, the semantics of the Node API are defined outside the JavaScript specification. Therefore, the V8 compilers cannot optimize the call trees in the two groups and the CPU times spent in the groups, which are up to 22.5% and 15.4% on average of the total CPU time.

Fig. 8 shows the CPU times spent in the Node and CalledByNode groups (the percentages are the ratios to the total CPU time). There are two major root non-JavaScript functions, node::StreamWrap::Writev() and node::Parser::Execute(). The two Node native functions further call the libv8 functions to access the properties of JavaScript objects. The two sub bars at the bottom in Fig. 8 show the CPU times while calling the two Node native code.

The CPU times spent in the Node and CalledByNode groups can be critical to the performance of server-side JavaScript frameworks that are developed by using non-JavaScript languages. For Node.js, the framework was developed by using C++ to use the V8 C++ interface. The server-side JavaScript frameworks need the function calls to the API of the JavaScript engines to handle JavaScript objects. The API functions are a part of the JavaScript engine code and are developed by using non-JavaScript in many cases (C++ for the V8 engine). We showed that more than 1,600 functions of libv8 were executed in the server-side workloads and most of the functions consumed less than 1% in Section III-F1. The JavaScript compilers cannot address the overheads of many function calls and cannot optimize the function calls by using the runtime profile feedback.

For the workloads using MySQL (update-mysql-raw, db-mysql, and db-mysql-raw), the call tree whose root is node::Start() shows the high CPU consumption. In the call tree, most of the CPU times were spent in its sub trees whose roots are the native code of the Node.js package module for MySQL binding. The Node.js package modules are developed by the open-source community for Node.js application programmers to enhance the functionality of Node.js. The Node.js package modules can provide the native code. For the MySQL workloads, the C++ native code handled JavaScript objects for the results of the database queries by calling the C++ API of V8.

5) Runtime Group: The root non-JavaScript functions of the Runtime group are the API functions of the V8 JavaScript engine. The compiled JavaScript code calls the API functions of the JavaScript engine to handle JavaScript objects. We observed 70 root non-JavaScript functions in the Runtime group were executed. Fig. 9 shows the CPU breakdown per root non-JavaScript function in the Runtime group.

A majority of the CPU times spent in the group are due to the function calls to four root non-JavaScript functions from the compiled code for creating, accessing, and transforming JavaScript objects. The four sub bars at the bottom of Fig. 9 list the functions: Runtime_KeyedGetProperty(), Runtime_StringSplit(), Runtime_CreateObjectLiteral(), and Runtime_SetProperty().

In particular, the function calls to a top hot function for accessing JavaScript objects, Runtime_KeyedGetProperty() or Runtime_SetProperty(), affected the total CPU times. The function calls to Runtime_KeyedGetProperty() consumed up to 6.8% of the total CPU time (4.2% on average) and the function calls to Runtime_SetProperty() consumed up to 6.6% of the total CPU time (1.6% on average).

We investigated what type of JavaScript code performed the function calls to the two hot functions. The call stack information collected by the perf tool contains the code addresses of the call sites that performed the function calls to Runtime_KeyedGetProperty() or Runtime_SetProperty(). Also the V8 compiler can dump the log files for the compiled code. We mapped the collected call site addresses to the compiled code in the code log files. However, because of a problem in the code mapping, we could not locate all of the call sites in the code log files since no compiled code that corresponds to the call sites was found.

We show how the hot Runtime functions such as Runtime_KeyedGetProperty() or Runtime_SetProperty() were called from the compiled JavaScript code by using the plaintext workload as an example. For the workload, the JavaScript functions of the Node.js API for handling the network requests (IncomingMessage._addHeaderLine() of the Http API and Url.parse() of the URL API) called the hot Runtime functions. Since the CPU time spent in the function calls to the hot Runtime functions is not negligible (6.7% of the total time), we should optimize the function calls to improve the overall performance of the workload. The JavaScript code that resulted in the hot function calls can include the code structures that are hard to be optimized by the current V8 compilers. In that case, if we can modify the source code of the JavaScript code so the compilers can remove the hot function calls, every application program based on the server framework benefits from reduction in the hot function calls since the JavaScript functions are a part of the server framework.

IV. Related work

To the best of our knowledge, there is little study on the characteristics of the servers-side JavaScript workloads. Richards et al. [14] analyzed how the dynamic features of the JavaScript language are used for the client-side JavaScript programs in the real world. They analyzed the language-level behaviors but did not address the characteristics of the compiled code.

Tiwari and Solihin [13] have performed the micro-architectural analysis of the dynamic behaviors for the client-side JavaScript benchmarks with the V8 engine. They focused on the characterization of the benchmark code in the hardware architecture level such as instruction parallelism, branch

Fig. 8. CPU time breakdown per root non-JavaScript function for the Node and CalledByNode groups

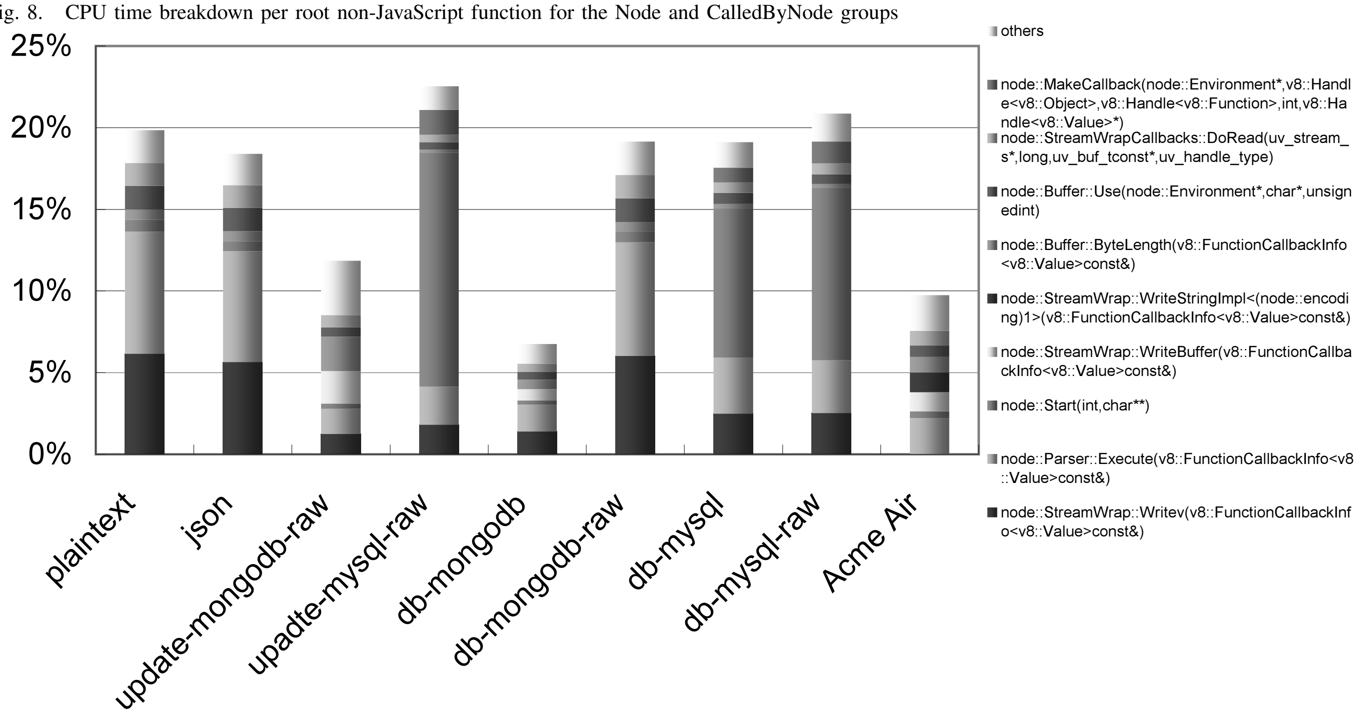
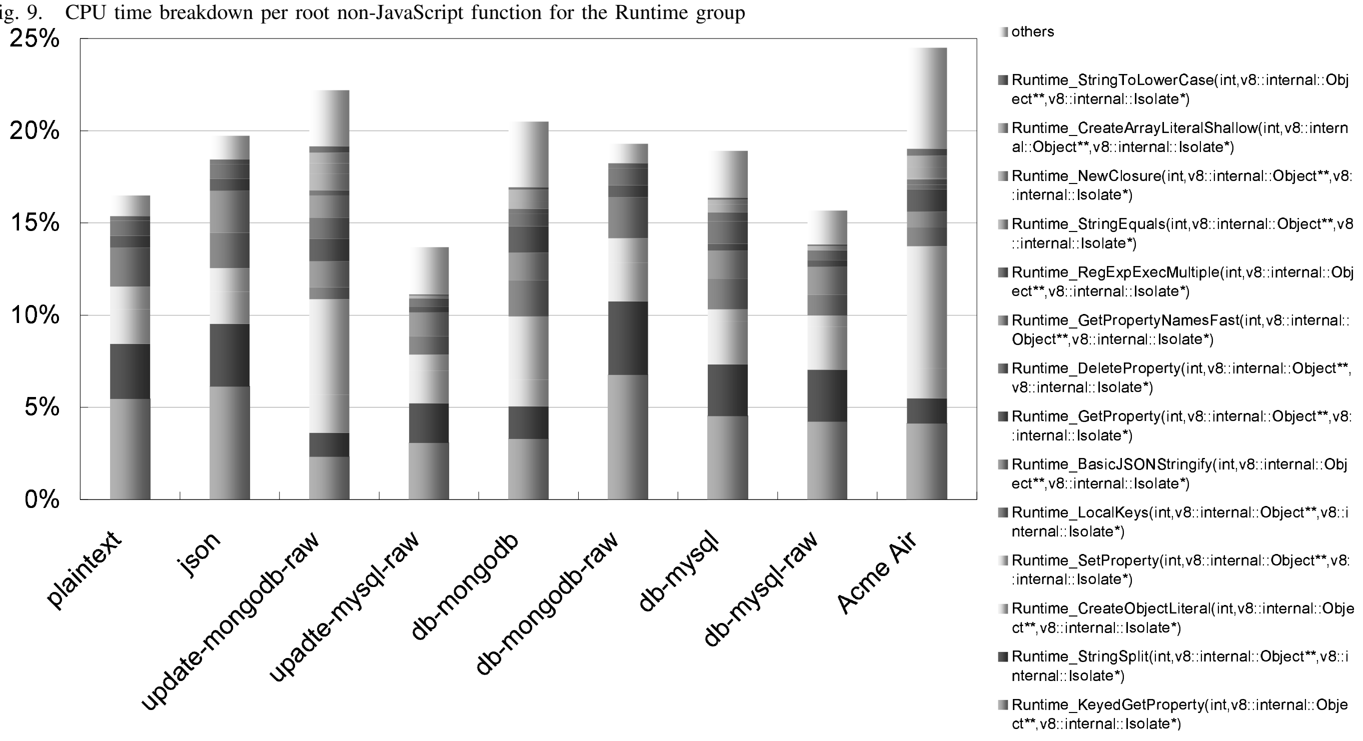


Fig. 9. CPU time breakdown per root non-JavaScript function for the Runtime group



prediction, cache hits, and so on. They did not address the server-side workloads.

For the dynamic typing languages including JavaScript, generating the type-specialized code can improve the performance of the programs. Kedlaya et al. [20] proposed a technique of combining type-feedback and type-inference to reduce the overhead of profiling and the cost of runtime guards while improving the accuracy of types. They reported that their technique can increase the number of type-inferred variables for the client-side programs.

V. Conclusions

For an emerging server-side JavaScript framework, Node.js with the V8 JavaScript engine, we demonstrated that the compilers currently target only 32.5% of the execution times for the server-side workloads such as Web Framework Bench and Acme Air application. The same compilers can target more than 80% (up to 98%) of the execution times and has significantly improved the performance for the client-side programs. The compiler optimization does not optimize the server-side workloads as it does for the client-side workloads.

We focused on the excessive CPU usage in the V8 library code and performed the detailed analysis of the call stacks when the V8 library code was executed. One reason for the excessive CPU usage in the V8 library is the function calls to the C++ code of the server-side JavaScript framework, which implements the Node.js API, and the native code in the Node Package Modules. The CPU usage caused by the function calls to the C++ server framework significantly affected the program execution time (up to 22.5% and 15.4% on average). The CPU usage will be hard to be optimized by the V8 compilers, since the Node.js API semantics is outside the JavaScript language.

Another reason is the function calls to the V8 runtime from the compiled JavaScript code. For the function calls, the V8 compilers have a chance of optimization. The compilers can understand the semantics of the function calls since the called functions are a part of the JavaScript engine. If it is hard for the compilers to optimize the function calls, and if modifying the JavaScript source code of the framework can reduce the function calls, many server applications will benefit from the changes of the framework code, since the CPU usage in the function calls is high (up to 24.5% and 18.7% on average).

Many functions of the V8 library were called for the server-side workloads. Most of the functions consumed less than 1% of the program execution time. The programs should suffer from the overheads of many function calls, however, the current V8 compilers cannot optimize the C++ call graphs. Also the compilers cannot use dynamic optimization based on the runtime type feedback for the C++ V8 runtime code. Therefore, the CPU times spent in the C++ call graphs remain as unoptimized while the compiler optimizes the JavaScript code. This problem should be commonly observed for any server frameworks for other scripting languages that are developed in C/C++ languages.

References

[1] JavaScript^{TM}. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[2] Python^{TM}. [Online]. Available: http://www.python.org/

[3] Google Inc. V8 JavaScript engine. [Online]. Available: https://code.google.com/p/v8/

[4] Node.js^{TM}. [Online]. Available: http://nodejs.org/

[5] N. Mostafa, C. Krintz, C. Cascaval, D. Edelsohn, P. Nagpurkar, and P. Wu, "Understanding the potential of interpreter-based optimizations for python," UCSB, Tech. Rep. Technical Report 2010-14, 2010.

[6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition (1st ed.).* Addison-Wesley Professional, 2013.

[7] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Jan. 1984.

[8] K. Ishizaki, T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani, "Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12).* New York, NY, USA: ACM, 2012, pp. 169–180.

[9] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java just-in-time compiler," *IBM Syst. J.*, vol. 39, no. 1, pp. 175–193, Jan. 2000.

[10] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline cachings," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91).* London, UK, UK: Springer-Verlag, 1991, pp. 21–38.

[11] J. Ansel, P. Marchenko, lfar Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*, 2011.

[12] V8 benchmark suite. [Online]. Available: http://v8.googlecode.com/svn/data/benchmarks/v7/run.html

[13] D. Tiwari and Y. Solihin, "Architectural characterization and similarity analysis of Sunspider and Google's V8 Javascript benchmarks," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12).* Washington, DC, USA: IEEE Computer Society, 2012, pp. 221–232.

[14] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI '10).* New York, NY, USA: ACM, 2010, pp. 1–12.

[15] V8: a tale of two compilers. [Online]. Available: http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers

[16] TechEmpower. Framework Benchmarks. [Online]. Available: http://www.techempower.com/benchmarks

[17] MongoDB. [Online]. Available: http://www.mongodb.org/

[18] IBM. Acme Air sample and benchmark. [Online]. Available: https://github.com/acmeair/acmeair

[19] jMeter workload instructions. [Online]. Available: https://github.com/acmeair/acmeair/wiki/jMeter-Workload-Instructions

[20] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf, "Improved type specialization for dynamic scripting languages," in *Proceedings of the 9th symposium on Dynamic languages (DLS '13).* New York, NY, USA: ACM, 2013, pp. 37–48.