2015

Architectural Characterization of Client-side JavaScript Workloads & Analysis of Software Optimizations

Malek Musleh
Purdue University, musleh@purdue.edu

Vijay S. Pai
Purdue, vpai@purdue.edu

Architectural Characterization of Client-side JavaScript Workloads & Analysis of Software Optimizations

Malek Musleh
School of Electrical and
Computer Engineering
Purdue University
West Lafayette, IN 47907
Email: musleh@purdue.edu

Vijay Pai
Purdue University, Google Inc
Mountainview, CA
Email: vpai@purdue.edu

***Abstract*—The use of JavaScript for web applications has increased in recent years due to its short learning curve, maintainability, and ease of portability across different mobile operating systems. However, the performance of dynamic-typed languages such as JavaScript significantly lag behind their corresponding C/C++ native code. Recent advances in JavaScript compilers have resulted in notable improvements for client-side Web applications. However, the impact on the architectural execution of these software optimizations is not well understood. Furthermore, because many of the client-side benchmarks are not representative of real world websites and applications, thus the would-be impact of many software optimizations may be overestimated.**

In this paper, we present an in-depth architectural characterization of a newer JavaScript benchmark suite, as well as a profile analysis of the popular JavaScript v8 runtime engine from Google. Our analysis illustrates the architectural impact of specific runtime optimizations, as well as additional insights of the interaction between the runtime engine and different JavaScript applications. We perform our evaluations using stock versions of new JetStream client-side JavaScript workloads, as well as on modified versions that closes the program-level representation gap with real-world websites.

I. Introduction

The emergence of the smartphone and tablet devices as widely used consumer devices has focused the community's attention on improving web-based applications. These applications are typically written in Javascript as programmers need to balance users' demand for rich graphical interfaces and support a large range of user inputs, along with their own desire to provide portability, easier maintenance, and fast development time across a range of mobile platforms. Dynamic typed languages reap such benefits through the facilitation of a high-level abstraction, but come at the cost of slower performance when compared to procedural and object-oriented languages such as C, C++. Despite recent innovations towards increased performance and power-efficiency in mobile platforms, embedded processors do not employ some of the more advanced architectural techniques (e.g. deep pipelines, complex branch predictors) seen on desktop and server platforms because of power and space constraints.

Furthermore, Ratanaworabhan et al. show that existing JavaScript benchmark suites such as Octane [1], Sunspider [2], Emscripten [3], and V8 [4] have been shown to characteristically differ from real-world websites and modeling of user-activity [5]. More specifically, they observe that real web applications contain much more JavaScript code than compared to V8 and SunSpider, with most of it being *cold* as it is never invoked. To bridge this representation gap, they show that the benchmark results containing 1MB of coldcode are more likely to be representative of browser performance on real web sites. This representation gap has hindered the community's ability to truly understand the complexity performance tradeoffs of advanced architectural optimizations.

In addition, understanding the execution characteristics of the runtime engine, as well as its impact on the system architecture provides yet another dimension towards bridging the performance gap. Maximizing performance requires minimizing runtime execution time. and thus a deeper understanding of the performance implications caused by runtime engine's interaction with the JavaScript workload can enable programmers to better identify program-level bottlenecks.

To help address these challenges, several previous papers have performed characterization studies: such as analyzing the architectural impact of conventional JavaScript the SunSpider and V8 benchmarks [6], analyzing the program-level difference between these conventional benchmarks and real websites, as well as suggesting how the program representation gap can be minimized [5]. Others have done a large scale study of the use of Eval in JavaScript applications [7], as well as a limit study of JavaScript parallelism [8].

In this paper we perform both an architectural and runtime evaluation of client-side JavaScript (JS) applications in the context of mobile platforms. Our detailed analysis shows the impact, and in many cases high variability of runtime characteristics that call into question the applicability of traditional optimizations geared towards applicability of a broader set of workloads. We make the following contributions:

- Architectural characterization of stock Mobile & JetStream suites
- Examination of architectural and performance impact of modified JS workloads intended to bridge the representation gap

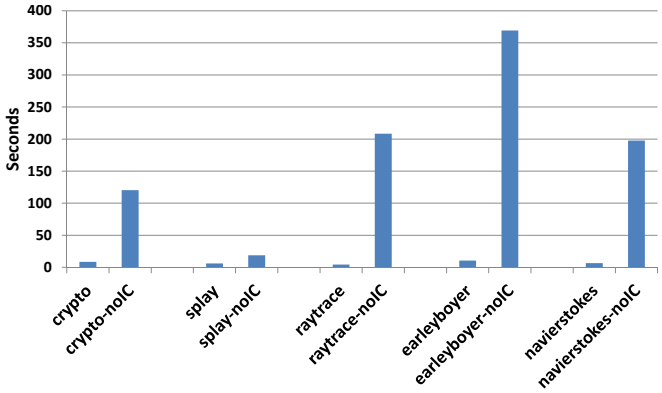


Fig. 1: InlineCache Impact on V8 Performance

- Program-phase variability behavior analysis
- Evaluation of the architectural impact of InlineCaches
- Performance analysis of V8 JavaScript runtime engine

The rest of this paper is organized as follows: Section II provides some background discussion and for this work. Section III discusses the methodology for our experiments. We discuss the architectural characterization in Section IV, and our runtime characterization in Section V. We cover related work in Section VI, and conclude in Section VII.

II. Background

A. Javascript Runtime Engine

1) Overview: In this paper, we specifically consider the open-source V8 Javascript Engine [4]. V8 consists of two compilers: the *full* compiler that is responsible for generating (inefficient) code quickly as possible; and an optimizing compiler, *Crankshaft*, responsible for generating optimized code for *hot code*, code that gets executed frequently. Crankshaft performs type analysis through the use of *Inline Caches* to refine its knowledge of program types so that efficient code can be generated.

2) Inline Caches: Inline Caching (IC) is a commonly implemented technique used for improving performance of dynamic-typed langauges [9]. A key assumption of dynamic-typed langauges is that property accesses at a given access site are usually performed on objects with the same type. Thus, maintaining type information regarding object properties provides a *fast path* for execution, without requiring invoking the costly runtime system to determine an object's property type.

It is well established that the use of ICs significantly improve performance [10]. Figure 1 shows the impact on performance (lower is better) of having ICs-enabled versus disabled for several of the V8 benchmarks. The X-axis denotes the benchmark name, and the Y-axis its corresponding execution time. X-axis labels appended with *no-IC* represent the benchmark having ICs disabled. We later show an analysis breakdown of the architectural characterstics of ICs in Section V-A.

As illustrated, enabling IC can provide up to orders of magnitude better performance as a result of efficient type-predictability. However, their effectiveness for these JavaScript suites cannnot be extrapolated to their actual impact for real websites. As others have shown, the behavior of JavaScript code is more dynamic in real websites, such that type-predictability is significantly harder [5], [7].

B. Architectural Characteristics

Recent works provide detailed micro-architectural characterizations of mobile and event-driven applications, and show that they are significantly different than widely used CPU-intensive/desktop benchmarks [6], [11], [12]. More specifically, these workloads suffer from significantly higher instruction cache misses and branch mispredictions - due to increased code size as well as extensive time spent in shared libraries and operating system processes. Moreover, the execution switches among different binaries frequently, such that they are interleaved rather than structured into discrete phases. In such situations, instruction locality and branch prediction accuracy may be affected, resulting in poor performance. Furthermore, the event-driven nature of web applications adds another level of obscurity to predicting program control flow.

III. Experimental Methodology

A. Workloads

For this work, we use benchmarks from the JetStream suite, [13], the Massive benchmark [14], BBench [11], and MobyBench [15]. The JetStream suite combines a variety of benchmarks that cover a variety of advanced workloads and programming techniques. It includes benchmarks from SunSpider 1.0.2 [2], Octane 2 [1], and Emscripten 1.13 [3].

Although JetStream introduces several upgrades (e.g. increased working set sizes, bug fixes) to the standard JavaScript suites of SunSpider and Octane, most of their tests remain small. Although some of the JetStream benchmarks test asm.js performance, several aspects are not fully measured. Specifically, asm.js often appears as large files, which offer different performance characteristics than that of smaller source files.

Large functions are difficult for the JavaScript runtime engine to optimize efficiently due to their sheer size. Because large codebases are likely to contain very large functions, these functions are not fully optimized. In our analysis, the Massive benchmark is included in order to stress test the memory subsystem given a larger code footprint. MobyBench is a benchmark suite composed of mobile applications such as gaming, social networking, email, and audio/video players.

B. System

For our architectural characterization analysis we use the gem5 simulator [16]. The architecture simulated is based on a mobile system, running Android ICS version [17]. The architecture details are detailed in Table I. In contrast to modern systems, the default instruction cache in gem5 is coherent.

For our V8 runtime analysis, we use PIN [18] to extract the following information: opcode distribution, function call count, and dynamic function body size. PIN is used for this part of the analysis in order to avoid the simulation cost overhead and kernel rebuilding process for process and function name tracing required for gem5. The host system parameters used

for the PIN analysis is listed in Table II. Table III lists the JetStream benchmarks and Table IV benchmarks evaluated in this work.

Processor	1 (1 GHz) OoO ARM ISA
L1 (I and D) caches	each 32 KB 2-way, 32 byte-block, 2-cycle
Shared L2 cache	2 MB, 16-way unified, 32-byte blocks, 30-cycle
Main Memory	DDR3 256MB
Memory Bus	500MHz 12.8 GB/s
Kernel	Armdroid 2.6.35

TABLE I: Simulation System Parameters

Processor	4 (2 GHz) OoO x86 ISA
L1 (I and D) private caches	32 KB 2-way
Shared L2 cache	2 MB
Main Memory	12GB
Kernel	Linux-3.2.0-24-generic

TABLE II: Host System Parameters

IV. Architectural Characterization

In this section we focus our analysis on memory behavior of the applications in order to determine what types of architectural optimizations are likely to significantly improve workload performance.

A. Instruction & Data Footprints

Conventional scientific workloads exhibit an instruction footprint that is able to fit within the capacity of the L1 instruction cache, while the occupancy of the shared last-level cache (SLCC) is predominately used for data. In contrast, commericial workloads exhibit instruction footprints larger than the L1I cache, and thus depend on the SLCC for additional storage. However, for commercial workloads the instruction occupancy requirement is significantly smaller than its data counterpart given that the capacity of the SLCC in high-end servers is much larger than mobile platforms. Therefore, SLLC optimizations for commercial workloads have primarily focused on the data portion. In contrast, mobile platforms exhibit both large instruction footprints and a limited capacity SLLC such that an efficient management policy requires addressing both components. Figure 2 illustrates a breakdown of L2 accesses between instruction and data for several benchmarks. The horizontal axis represents the ratio, while the vertical axis denotes the benchmark. For each benchmark the ratio of accesses for the instruction, data, instruction TLB walker, and data TLB walker are shown.

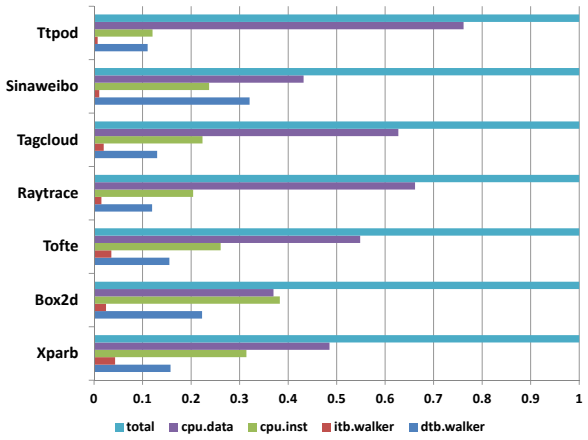


Fig. 2: **L2 Accesses Breakdown**

Benchmark	Category	suite
base64	Base Encoder/Decoder	SunSpider
bigfib	Fibonacci numbers	Emscripten
box2d	Physics Engine	Octane2
code-firstload	Code load speed	Octane2
cordic	Cordic Algoritm	SunSpider
crypto-aes	Advanced Encryption Standard	SunSpider
crypto-md5	MD5 Implementation	SunSpider
dry	Dhrystone synthetic computing	
earleyboyer	Chart Parser & Logic Programming	Octane2
gbemu	GameBoy emulator	Octane2
floatmm	Floating point matrix multiply	Emscripten
mandreel	Physics engine's performance	Octane2
navierstokes	Fluid simulation	Octane2
n-body	Solar system simulation	Sunspider
raytrace	Simple raytracer	SunSpider
regexp-2010	Regular expressions	Octane2
splay	Manipulation of Splay tree	Octane2
tagcloud	Parses JSON	SunSpider
tofte	Eval usage	SunSpider
typescript	Microsoft's Compiler	Octane2
xparb	Date Formatting/Parsing	SunSpider
zlib	Zlib compilation	Emscripten
3dcube	3d Cube rotation	SunSpider

TABLE III: JetStream Benchmarks

B. DeadBlock Analysis

Deadblock (blocks that are only used once) prediction can be used to identify blocks to be dead in order to drive cache management policies that replace deadblocks with live blocks (blocks with higher reuse). Improved cache policies will improve the efficiency and utilization of the caches increasing the hit rate, and thus leading to performance improvement.

Observation The deadblock ratio (the ratio of deadblocks to total blocks in the cache) varies significantly not only across different JavaScript workloads, but also between the instruction & data streams.

We breakdown the deadblock analysis results as follows. We first categorize deadblocks as either being instruction or data deadblocks. We also illustrate the deadblock ratio for

Benchmark	Category	Operation
360buy	Online Shopping	Lists products items
frozen_bubble	Game	Loads a game
kingsoftoffice	Document	Opens document file
netease	News	loads news content
sinaweibo	Social Network	Loads Information
ttpod	Audio	Plays a song

TABLE IV: MobyBench Suite

cache misses (the percentage of cache misses to blocks that are only used once). These results are illustrated in Table V. The significant deadblock miss ratio is noteworthy, as it suggests an opportunity for bypassing mechanisms to improve efficiency.

However, one point to be made is that predicting deadblocks for loop-based programs using techniques such as sampling the reference count are effective because control-flow access behavior is more predictable. In contrast, because event-driven web applications exhibit poor instruction locality and lower branch prediction accuracy due to the fact different sets of events are triggered depending on input and typically short executing body event-handlers, conventional techniques may not be as effective. Furthermore, deadblock predictors that utilize program counter (pc) such as [19] to drive bypassing also may not suitable in the presence of frequently occuring self-modifying code. This is because the optimizing and/or deoptimizing of JavaScript code will likely make it harder for such prediction schemes to maintain steady-state.

C. ColdCode-Effect

Previous work has shown that workloads from existing Javascript suites differ from their real-world counterparts [5], [20]. A main difference is that real web applications contain significantly more Javascript code, with most of it being *cold*, code that is processed by the runtime engine but never actually invoked during execution of the benchmark. To bridge this difference, we follow their example and add cold code to the benchmark suite in order to make them more representative and determine the resulting architectural characteristics. Specifically, we add about 1MB of JavaScript from the JQuery Mobile library as part of the input source code. However, functions from this library are never invoked by the workload and are thus only processed once by the JavaScript runtime during the loading phase.

The effects of the coldcode addition are illustrated in Figures 3 - 4. Figure 3 depicts the change in missrate while Figure 4 presents the change in overall access behavior for the instruction cache, data cache, and SLCC. The figures illustrate a grouped bar chart denoting the percent change of several cache-related metrics for two workloads: base64, and navierstokes. The X-axis illustrate the percent change (original version / coldcode verion) and the Y-axis indicates the metric.

Focusing on the top-level (instruction & data) cache missrate behavior in Figure 3, we see that the addition of the coldcode results in about 20-30% worsening of the instruction missrate, while about 15% worsening for the data cache missrate.

Benchmark	Instruction Ratio	Data Ratio
bigfib	0.33	0.13
box2d	0.24	0.45
cordic	0.31	0.50
codefirstload	0.48	0.42
dry	0.31	0.33
float	0.34	0.35
navierstokes	0.23	0.41
nbody-c	0.34	0.55
regexp-2010	0.22	0.42
typescript	0.24	0.36

TABLE V: DeadBlock Ratio

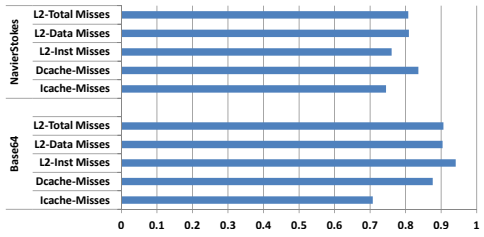


Fig. 3: **ColdCodeEffect on Misses**

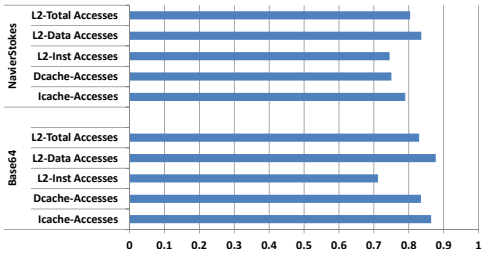


Fig. 4: **ColdCodeEffect on Accesses**

Observation The impact on the instruction cache is almost double the impact on the data cache due to the fact that because these workloads already exhibit high instruction misses per kilo-instruction (MPKI) due to their poor instruction locality, the addition of coldcode significantly reduces the amount of instruction reuse the benchmark was utilizing without the coldcode.

Turning our attention to the SLCC, we again see that the impact of coldcode is different for the L2-Inst missrate of NavierStokes than it is for Base64. Specifically, the L2-Inst missrate worsens more than compared to the L2-Data missrate for Navierstokes, whereas the opposite is true for Base64. This is likely due to the fact that because Base64 experiences greater instruction reuse (note the larger increase of L2-Inst Accesses for Base64 than NavierStokes (Figure 4), allowing the base LRU replacement policy to preserve the instructions.

Observation Despite the high instruction access to the SLCC, the overall missrate of the SLCC is closer linked to the L2-Data missrate component than that of the L2-Inst missrate. However, this does not suggest that a cache management policy that prefers data is likely to provide better performance as the effects of instruction & data misses are different.

D. IPC Distribution

In this subsection we evaluate the distribution of the sampled IPC (instructions per cycle) for the different workloads. The motivation for monitoring changes in the runtime IPC is to determine if a program exhibits significantly varying program phase changes, an observation that cannot be determined by simply looking at the average of the entire program. Figure 5 plots the sampled IPC into histograms for both the MobyBench and JetStream benchmarks. The plots illustrate that the JetStream benchmarks exhibits far greater variance between phases.

Observation This larger variance suggests that the JetStream benchmarks are much more likely to benefit from program phase optimizations, rather than through entire-program performance metrics. In contrast, the MobyBench suite illustrates comparable IPC performance throughout program exe-

cution, suggesting that overall program performance metrics are good indicators for performance tuners.

E. Memory Bandwidth

Figure 6 illustrates the memory bandwidth requirements for several of the benchmarks. As our target platform is that of a mobile platform, the parameters of the system are indicated in Table I. To maintain reasonable simulation times (1-2 days) the amount of simulated time is 30 seconds. We see that both the conventional (Sunspider, Octane), realistic (Box2d, Codeload), as well as mobile workloads (e.g. BBench) exhibit similar, yet underwhelming memory bandwidth demands.

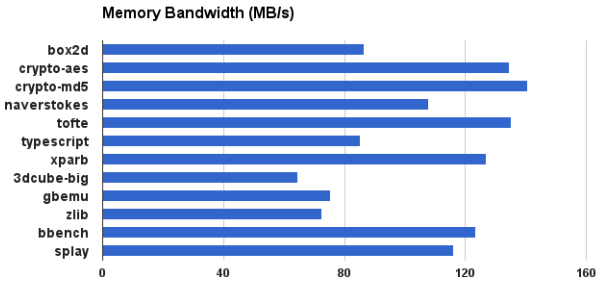


Fig. 6: **Memory Bandwidth**

V. V8 Runtime Characterization

In this section we discuss our performance profiling of the V8 JavaScript runtime engine. We analyze the runtime engine to collect profile analysis in order to identify functions that are invoked frequently and spend a lot of time in as areas for potential optimizations. Tiwari et al. [6] shows that a significant part of the total execution is spent in the V8 JavaScript runtime engine. Obtaining a deeper understanding of the bottlenecks of the runtime engine is required in order to be able to further optimize JavaScript performance.

In the following subsections we perform an instruction characterization of InlineCaches and identify *hot code* of V8. Characterizing the impact of ICs would suggest the type of processor support needed to improve performance (e.g. additional ALU units, bigger branch-history tables). Similarly, identifying the heavily used components of V8 can suggest opportunities for architectural enhancements unique to JavaScript performance.

A. Instruction Characterization of Inline Cache Performance

Because of the limited effectiveness of ICs in real websites, it is important to consider the architectural impact for when ICs are disabled. Specifically, we analyze several architectural metrics: Instruction Opcode distrbution and number of instructions. Figure 7 illustrates the architectural break the impact of ICs for several of the V8 benchmarks. For each of the subplots, the X-axis indicates the benchmark name, while the Y-axis represents the relative increase in operations. We observe that the impact of having ICs disabled results in significantly more instructions being executed. Subplot 7e illustrates the overall total operations increase of the experiments.

Observation One key observation we see that is for different benchmarks exhibit different increases for each type of instruction. For instance, raytrace experiences the largest relative increase of branch and Integer ALU operations, but NavierStokes sees the largest relative increase in memory operations. On the other hand, Crypto experiences the largest increase in comparison operations. This indicates that optimizations that target one opcode reduction (e.g. minimizing ALU operations) would not be effective for all benchmarks.

B. V8 Performance

In this subsection we show the performance characteristics of the V8 runtime engine when executing the different benchmarks. In addition to identifying *hot code* as areas for optimization, understanding the execution behavior of the runtime illustrates the specific demands of the JavaScript benchmark at a program-level. For instance, illustrating that a large increase SLCC misses is due to poor type-predictability that is identified by observing that a particular runtime function is invoked repeatedly would help JavaScript programmers tune their benchmarks accordingly.

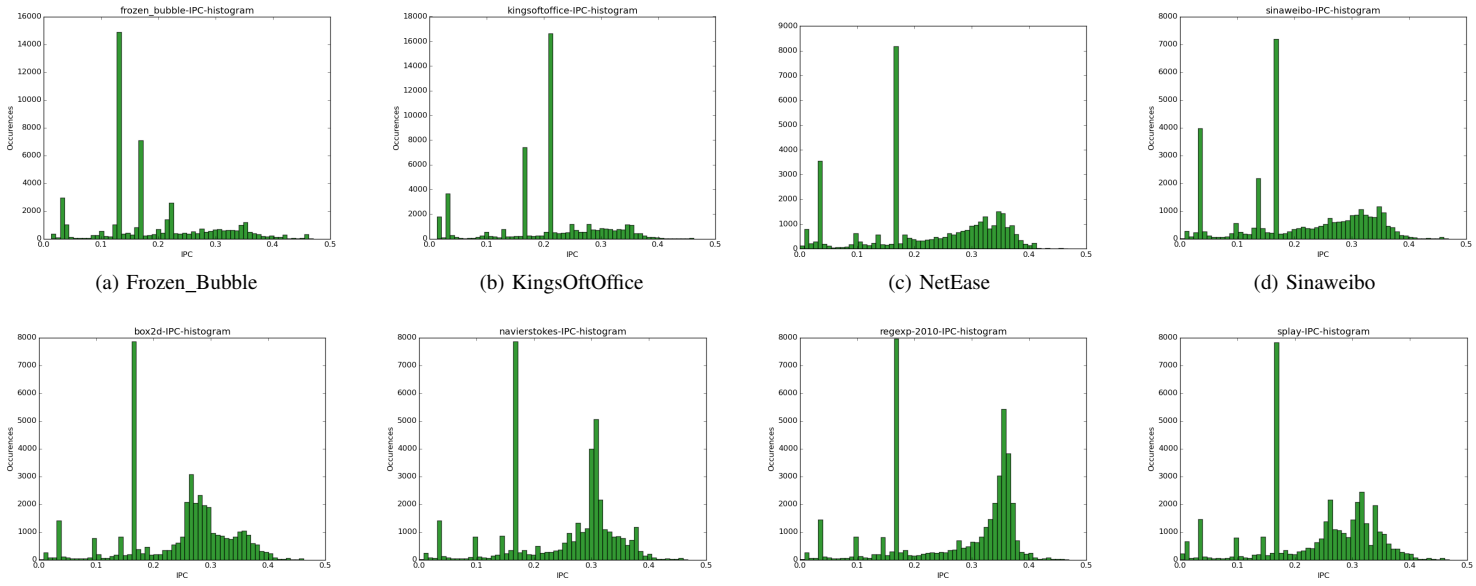
The results presented in this subsection have all the default runtime optimizations enabled (e.g. Inlining Cache). We tally the function call count, dynamic function body size, and their multiplcative factor to determine which parts are suitable for optimizations. However, when deciding what parts are good targets for optimizing, consideration is typically given to common bottlenecks across a set of workloads rather than benchmarks individually. This is done in order to determine what optimizations are more likely to benefit a broader target audience or domain.

For this we group all the workload function profiles together to determine which functions are invoked the most often, and then focus our attention on the top twenty functions. Figure 8 illustrates the top twenty functions that are called in consideration of the entire suite. The vertical axis indicates the function name, and the horizontal axis represents the number of times it was called.

We see that the top three functions are: *InternalLookupIterator, HandleScope, and FactoryNewNumber*. FactoryNewNumber corresponds to the allocation of new memory on the heap, thus indicating that these benchmarks are memory intensive. The other top 2 functions correspond to calls to the construction of the Iterator class. High usage of the iterator class suggests that a lot of time is spent traversing graphs, or other unordered map-type datastructures. An important example of this is the Global Value Numbering (GVN) optimization performaned by Crankshaft. GVN is an optimization that attempts to eliminate redundancy. More specifically, for each instruction it generates a hashcode based on its opcode, input, any data associated with it, and then inserts it into a hash table. When there exists an equivalent instruction in the hash table, GVN deletes the later instruction and uses the one already in the table [21].

The remaining top functions mainly correspond to runtime object property settings, which have to do with updates (SetProperty) and reads (getProperty) of dynamically allocated objects.

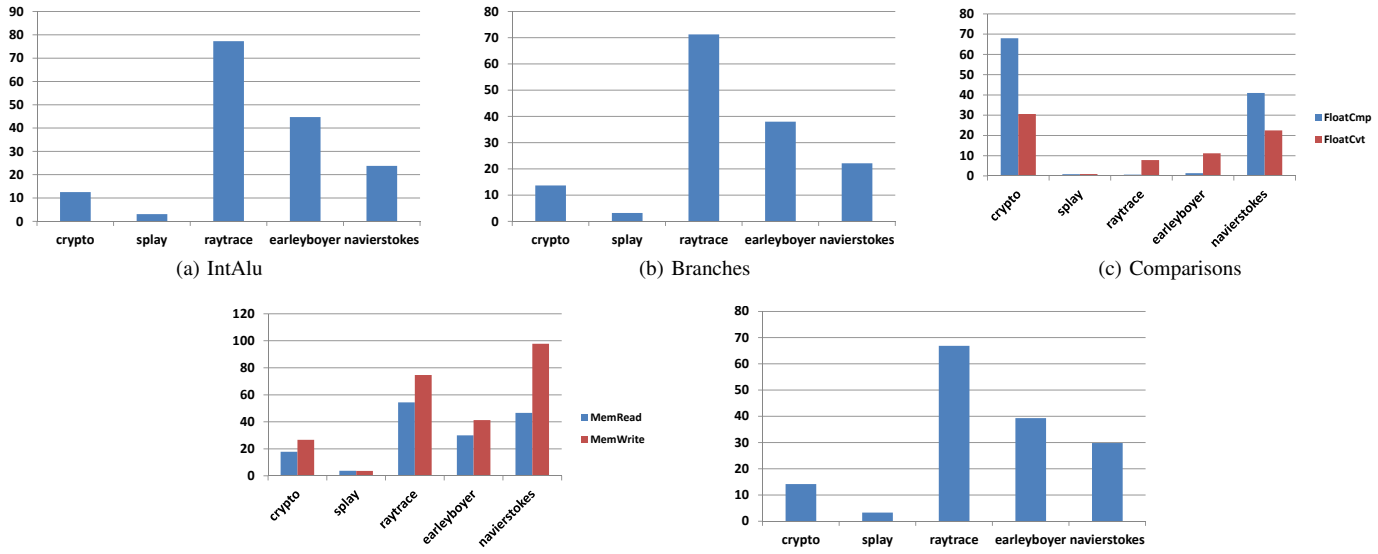
Figure 10 illustrates the V8 function profiles for several individual benchmarks. Viewing the fuction call distribution



(a) Frozen_Bubble (b) KingsOftOffice (c) NetEase (d) Sinaweibo

(e) Box2d (f) NavierStokes (g) Regexp-2010 (h) Splay

Fig. 5: **IPC Distribution for MobyBench & JetStream Benchmarks**



(a) IntAlu (b) Branches (c) Comparisons

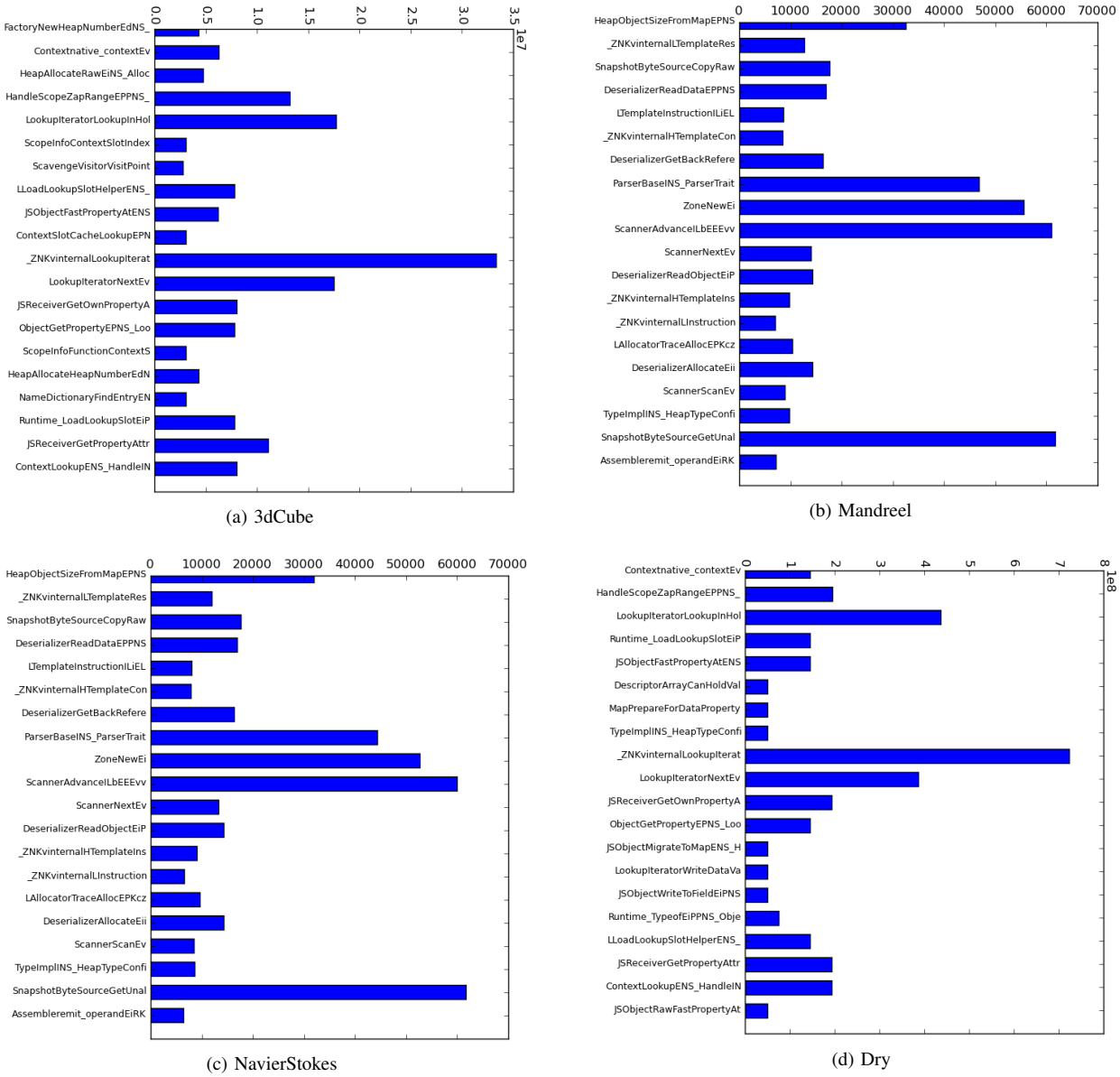
(d) MemOperations (e) TotalOperations

Fig. 7: **V8 InlineCache Impact Breakdown**

for the benchmarks individually help illustrate unique trends. Specifically, for computationally intensive workloads such as NavierStokes and Mandreel, we see that *Parser* and *ScannerAdvance* are invoked frequently. Furthermore, these two functions are invoked as frequently as the *LookUpIterator*. In contrast, *Parser* and *Scanner* are not listed in the graphs suggesting that optimizing those two functions are likely to only benefit computational intensive workloads.

Another observation from the individual benchmark profiles suggest that these benchmarks frequently allocate memory on the heap. This can probably be correlated back to how JavaScript programs are typically written: frequent allocation of memory for new objects with little concern for memory management. Despite the frequent memory allocation, we only see the garbage collector *Scavenger* routine show up in the 3dCube profile, suggesting that these benchmarks do not stress the memory-limits of the system and most objects allocated are predominately live.

After profiling the most invoked functions, we further investigate the impact of these functions on total V8 performance. More specifically, even though a function may be invoked many times, its *cost*, in terms of number of program instructions or cycles can be relatively minimal compared to other less frequently invoked *larger* functions. To better determine the true cost for each runtime function, we take the function call count and multiply it by the dynamic bodysize of the function. For simplication purposes we assume that all instructions have an equvalent cost, such that the larger the mutlplied factor, the greater its overall share of program execution time.



(a) 3dCube (b) Mandreel

(c) NavierStokes (d) Dry

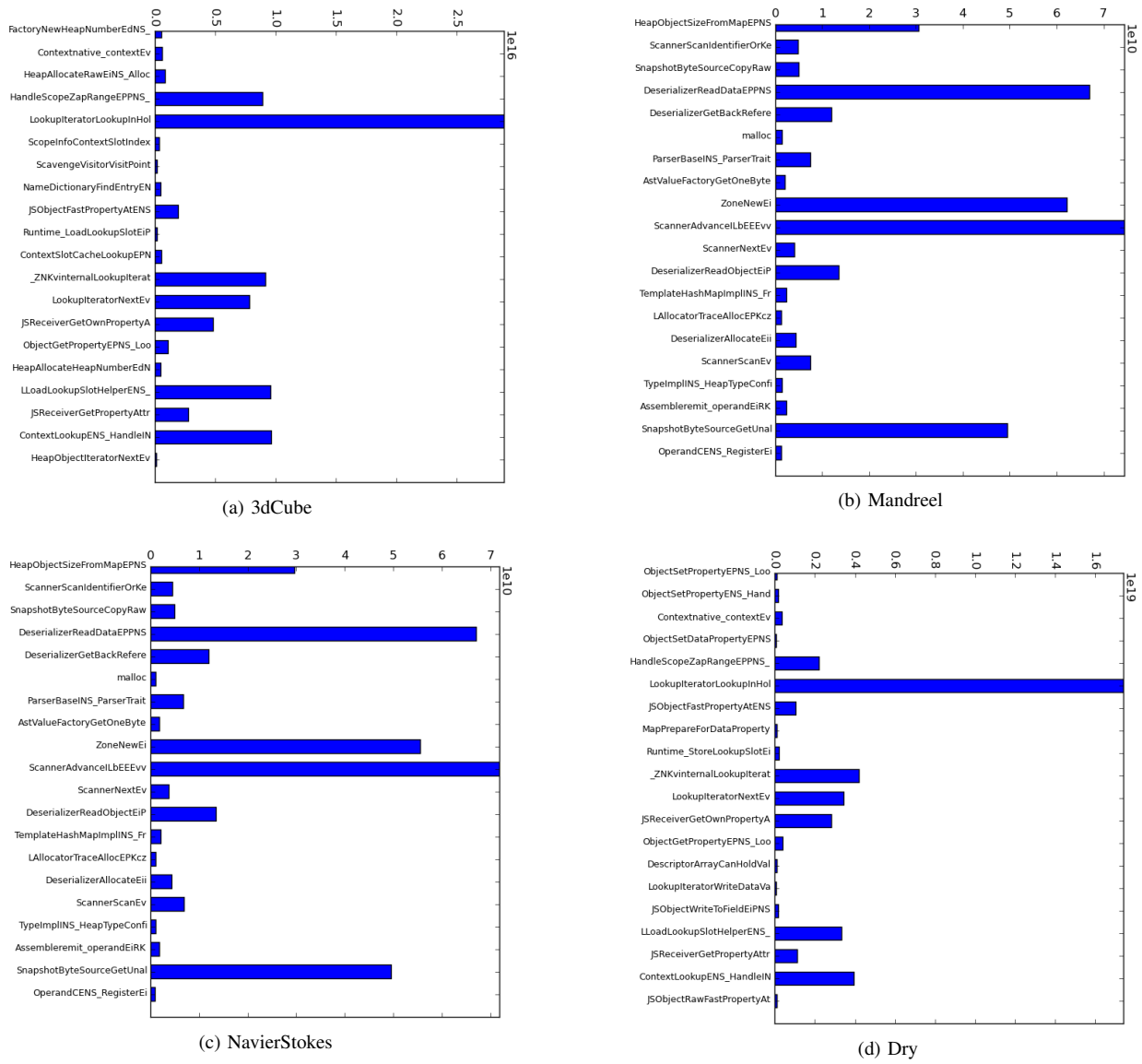
Fig. 9: **V8 RunTime Function Profile Benchmarks**
Illustration of the top twenty functions invoked for several benchmarks.

C. V8 Performance: IC-Disabled

In this subsection we evaluate the performance characteristics of the V8 runtime engine with Inline Caching disabled. Because the effectiveness of IC decreases for realistic webpage browsing, an evaluation of the runtime engine under this scenario is key to better understand areas for optimizing. Because of the increased execution time associated with disabling IC, we illustrate the comparable effects of enabling/disabling IC on the shorter executing V8 workloads. Figure 11 illustrate the function profile analysis on the runtime when enabling/disabling IC for the NavierStokes and Splay benchmarks. For each of the subplots the X-axis illustrates the top twenty invoked functions, while the Y-axis represents the call count.

Observation We see that the hot functions differ between the when ICs are enabled and when they are disabled. This suggests that optimizations targeted for this scenario may not necessarily be applicable or useful when ICs are effective. For instance, subfigure 11c shows that the dominant function calls for executing the Splay benchmark with IC enabled are memory-related: heap allocation, and the garbage collector (Scavenger). In contrast, disabling IC for the Splay benchmark (subfigure 11d) introduces multiple *LookupIterator* function calls as well as calls to *LoadIC_Miss* handling, the latter of which would be expected.

Observation Another key difference is the orders of magnitude difference in executed instructions noted in the Y-axis count between Figures 11a and 11b. Because IC does not have as much of an impact on splay's performance, the Y-axis counts between Figures 11c and 11d are the same. The execution of the benchmarks with ICs disabled result in the functions being invoked significantly more times, and hence



(a) 3dCube

(b) Mandreel

(c) NavierStokes

(d) Dry

Fig. 10: **V8 RunTime Function Total Cost Profile Benchmarks**
Illustration of the top twenty functions total cost for several benchmarks. The result for each function corresponds to its call count multiplied by its dynamic function body size.

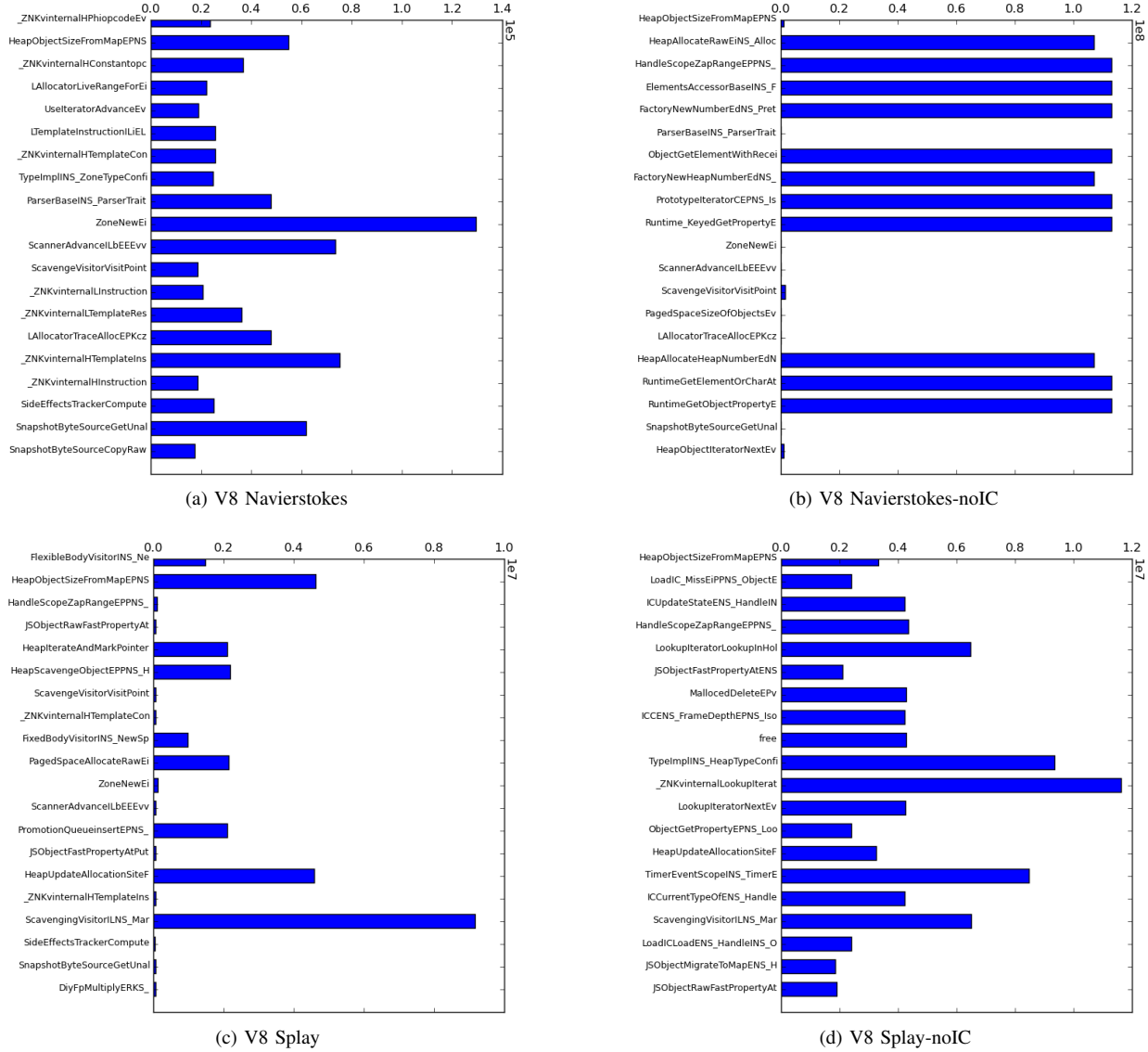
execute orders of magnitude more instructions than when ICs are enabled. This again correlates back to Figure 7 illustrating the large increase of number instructions executed.

VI. Related Work

Because of the renewed focus on improving mobile and web performance, recent proposals have dedicated their efforts to better understand JavaScript workload characteristics and propose optimizations for improving performance.

A. Analysis Studies

Fortuna et al. conducted a limited study on the potential of JavaScript parallelism [8]. Ogasawara focuses on the runtime analysis for server-side JavaScript workloads [10]. Tiwari et al. perform an architectural characterization and principle component similarity analysis of the SunSpider and V8 benchmarks [6]. Due to the representation gap between the standard JavaScript workloads and real websites' content and modeling of user-interaction, others have strived towards creating more realistic frameworks: Gutierrez et al. created the general web-browsing benchmark: BBench [11]. Richards et al. have developed an automated browsing benchmark, JS-Bench, that models a fixed period of user interaction for several of the most popular websites [20]. Pandiyan et al. performan energy characterization and architectural implication study of their own MobileBench suite [12]. They enhance BBench to incorporate more realistic user actions such as different scrolling & zooming movements. The primary focus of both the BBench and MobileBench works have been to determine the architectural and energy characterization of mobile workloads such as general web browsing, and loading of games or photos in a gallery. Due to the limited user-interaction modeling presented in their benchmarks, the JavaScript engine is



(a) V8 Navierstokes

(b) V8 Navierstokes-noIC

(c) V8 Splay

(d) V8 Splay-noIC

Fig. 11: **V8 RunTime Function Profile Inline Caching Effect**

barly utilized. Our analysis essentially covers both dimensions: architectural characterization of a newer JavaScript suite and a detailed performance analysis of a popular JavaScript runtime engine. Furthermore, our analysis is provided in the context of a mobile system which better illustrates the architectural impact of the runtimes' optimization & workload behavior in a more constrainted environment.

B. Proposed JavaScript Optimizations

EFETCH, RDIP, and PIF are Instruction-cache prefetchers have been recently proposed [22], [23], [24]. The first two exploit characteristics of event-driven workloads, to create an *event signature* to capture current event and function calling context to predict the control-flow within a function. There are other proposed techniques to improve JavaScript execution that are orthogonal to this work. Anderson et al. propose extending the ISA to load and check the type with a single instruction [25]. Ahn et al. identify two design decisions in the V8 JavaScript Engine that hinders type predictability. They propose three software optimizations that minimizes type unpredictability and reduces the dynamic instruction count for execution for real websites [26]. Zhu et. al perform design-space exploration to determine energy efficiencies trade-off between processor designs for web applications. They also propose implementing the browser engine cache as a collection of registers where each register holds exactly one DOM (render) tree attribute [27].

VII. Conclusion

The increased popularity of JavaScript enabled browsers and applications has focused the community's efforts towards improving performance. Given that its event-based programming model is inherently different than conventional scientific workloads, requires solutions for optimizing performance that ones that have been applicable to other domains. It is this observation that motivates the need to investigate the performance characteristics at a finer-grain level that has been done before. In this paper, we present an in-depth architectural

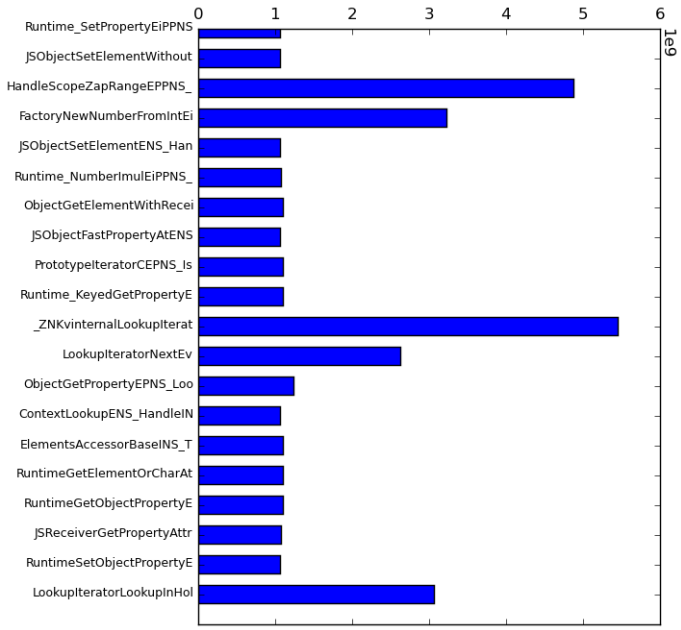


Fig. 8: **V8 Function Total Call Count**
Illustration of the top twenty functions invoked across the entire suite.

characterization of the new JetStream JavaScript suite. We investigate key characteristics such as deadblock ratio, coldcode-effect on cache performance, IPC variability, and memory bandwidth. We also examine the instruction characterization of type-predictability, and perform a detailed study of the interaction between the runtime engine. We find that tailoring stock JavaScript bencmarks towards more realistic web-sites significantly increases the system memory pressure. Moreover, we illustrate the variance in program-phase behavior, as well as in runtime behavior due to presence or absence of Inline Caching. This variance signifies that optimizations that target specific JavaScript program-phases or accelerating ineffective type-predictability are more likely to be impactful and useful than traditional optimizations that have been targeted for broad applicability.

REFERENCES

[1] Octane 2.0. [Online]. Available: https://developers.google.com/octane/

[2] Sunspider 1.0.2 javascript benchmark. [Online]. Available: https://www.webkit.org/perf/sunspider/sunspider.html

[3] Emscripten benchmark suite. [Online]. Available: http://kripken.github.io/embenchen/

[4] V8. [Online]. Available: https://v8.googlecode.com/svn/data/benchmarks/v7/run.html

[5] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, ser. WebApps'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.

[6] 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*, ser. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 221–232.

[7] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–78.

[8] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of javascript parallelism," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[9] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84. New York, NY, USA: ACM, 1984, pp. 297–302.

[10] T. Ogasawara, "Workload characterization of server-side javascript," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014, pp. 13–21.

[11] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *the proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 81–90.

[12] D. Pandiyan, S. Lee, and C. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench," in *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2013, Portland, OR, USA, September 22-24, 2013*, 2013, pp. 133–142.

[13] Jetstream benchmark suite. [Online]. Available: http://browserbench.org/JetStream/in-depth.html

[14] Massive benchmark. [Online]. Available: http://kripken.github.io/Massive/

[15] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 45–54.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhaȓdt, A. Saidi, A. Basu, J. Hestness, . D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewel̃l, M. Shoaib, N. Vaish, M. D. Hill, and D. i. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[17] Android ice cream sandwich. [Online]. Available: http:://www.android.com

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[19] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 175–186.

[20] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of javascript benchmarks," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 677–694.

[21] Jetstream benchmark suite. [Online]. Available: http://jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler

[22] G. Chadha, S. A. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven webapplications," in *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, 2014, pp. 75–86.

[23] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: Return-address-stack directed instruction prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 260–271.

[24] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 152–162.

[25] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked load: Architectural support for javascript type-checking on mobile processors," in *Proceedings of the 2011 IEEE 17th International Symposium on High*

Performance Computer Architecture, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 419–430.

[26] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving javascript performance by deconstructing the type system," *SIGPLAN Not.*, vol. 49, no. 6, pp. 496–507, Jun. 2014.

[27] Y. Zhu and V. J. Reddi, "Webcore: Architectural support for mobileweb browsing," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 541–552, Jun. 2014.