Architectural Characterization and Similarity Analysis of Sunspider and Google's V8 Javascript Benchmarks *

Devesh Tiwari and Yan Solihin
Department of Electrical and Computer Engineering
North Carolina State University
{devesh.dtiwari,solihin}@ncsu.edu

ABSTRACT

Today, more than 99% of web-browsers are enabled with Javascript capabilities, and Javascript's popularity is only going to increase in the future. However, due to bytecode interpretation, Javascript codes suffer from severe performance penalty (up to 50x slower) compared to the corresponding native C/C++ code. We recognize that the first step to bridge this performance gap is to understand the the architectural execution characteristics of Javascript benchmarks. Therefore, this paper presents an in-depth architectural characterization of widely used V8 and Sunspider Javascript benchmarks using Google's V8 javascript engine. Using statistical data analysis techniques, our characterization study discovers and explains correlation among different execution characteristics in microarchitecture dependent as well as microarchitecture independent fashion. Furthermore, our study measures (dis)similarity among 33 different Javascript benchmarks and discusses its implications. Given the widespread use of Javascripts, we believe our findings are useful for both performance analysis and benchmarking communities.

1. INTRODUCTION

The wide adoption and usage of Javascript is well known [4]. More than 95% of online users experience web browsing enabled with Javascript, and that is not a surprise because more than 99% web sites use Javascripts [4]. The primary reason for Javascript's popularity is ease of development, deployment, and portability which are facilitated by its high level of abstraction.

High level abstraction has acted as a catalyst for Javascript's wide adoption but at the cost of a severe performance penalty. Recent studies show that Javascript implementations can be up to 50x slower than corresponding C/C++ implementations [24]. This performance gap is mainly due to bytecode interpretation of dynamically typed Javascript code as opposed to native code generation for statically typed languages C/C++. As Javascript applications become more complex in the future, this performance gap is expected to further increase [24].

The first step to bridge this performance gap requires understanding the architectural characteristics of Javascript applications. Though there have been previous works optimizing widely used V8 [3] and Sunspider [2] Javascript benchmarks [6, 12], architectural execution characteristics of these benchmarks remain uninvestigated. Understanding and analyzing architectural execution characteristics is critical for system designers looking to mitigate performance bottlenecks of future computing devices (e.g. handheld mobile platforms) [11]. Given these Javascript benchmarks are being widely used for variety of purposes including performance ranking of web browsers [41, 42], it becomes necessary to develop such an understanding. Therefore, *the goal of this paper is to understand and analyze architectural execution characteristics of Javascript benchmarks.*

Unfortunately, characterizing workloads rigorously is challenging for many reasons. Typically, system designers use architecture simulator to characterize or to find out bottlenecks for a given workload. Though robust, due to the inherent slow nature of simulators, this approach suffers from excessive slowdown, limiting its practicality [22]. Another approach is to characterize the workload on a given processor architecture to gain first hand insights. This approach is relatively less time consuming and hence, has been used widely in many previous characterization studies [9, 13, 30, 37]. Though fast and widely used, this approach is limited in scope due to its dependence on the underlying processor architecture.

Not only collecting the characterization data is time consuming and difficult, but analyzing the characterization data is challenging as well. The amount of the data generated by characterization techniques is often so huge that meaningful data analysis becomes very difficult. Furthermore, execution characteristics often interact with each other in a complex manner, and these complex interactions often prohibit us from drawing statistically meaningful conclusions. In summary, a generic characterization approach needs to meet these challenges and be able to provide statistically meaningful insights.

To address these challenges, this paper applies microarchitecture dependent as well as micro-architecture independent (program inherent) characterization methodology. Microarchitecture dependent characterization approach leverages hardware performance counters to understand various execution metrics. Our hardware counter based characterization provides fast first hand insights, but by definition it may depend on the underlying processor architecture. To overcome this limitation, we then use microarchitecture independent characteristics such as ILP, control-flow predictability, instruction and data locality, etc. to gain additional insights. Though slower than hardware counter based characterization, it is orders of magnitude faster than architectural simulation. We also compare and contrast these two characterization approaches, and discuss the implications of those for performance analysts and future processor designers.

These characterization approaches produce huge data that needs to be analyzed. To alleviate the analysis challenge, we apply multivariate statistical data analysis technique, principal component analysis (PCA), to reduce the dimensionality of measured execution characteristics while retaining most of the information. Using PCA, we reduce the data size to be analyzed without loosing much information and still maintaining high statistical confidence. This analysis enables us to quantify the correlation and interaction among different execution characteristics in statistically meaningful way. In addition to these techniques, we employ agglomerative clustering analysis to investigate similarity across different Javascript benchmarks, and discuss the implications of these for future processor designs.

Studying the similarity across emerging workloads is critical be-

*This work was supported in part by NSF Award CNS-0834664. Authors would like to thank anonymous reviewers for their helpful feedback.

cause it helps in selecting a subset of benchmarks that is still representative of the original target workload [28], facilitating faster simulation time for new architectural techniques, faster design space exploration for future processor designs [18], etc. With ever growing target workload space, the need for careful selection of representative benchmarks with minimal redundancy will only continue to grow.

This work makes following contributions:

- To the best of our knowledge, this is the first in-depth architectural execution characterization of widely used V8 and Sunspider Javascript benchmarks.
- We find that V8 benchmarks have high instruction level parallelism despite the high branch frequency. On the other hand, Sunspider benchmarks encounter branches less frequently but suffer from a higher branch misprediction penalty, resulting in more wasteful work and hence, limiting its overall IPC.
- Our study finds that Sunspider benchmarks have rate of higher cache accesses, TLB misses, and higher average L1 cache access latency compared to V8 benchmarks. However, V8 benchmarks have higher rate of L2 cache and memory bandwidth requirements.
- Our analysis discovers that there is significant redundancy among Javascript benchmarks. Sunspider benchmarks exhibit significant similarity for wide range of execution characteristics, though Google's V8 benchmarks are diverse in their architectural execution characteristics.
- Our study shows why joint exploration of hardware counter based characterization and micro-architecture independent characterization is necessary for identifying performance bottlenecks. We also make a case for the need to provide a wider variety of hardware counters in future processors.
- We also compare SPEC 2006 benchmarks and Javascript benchmarks using microarchitecture independent characteristics and show why adding some of the Javascript benchmarks will be useful for benchmarking and guiding future processor designs.

With rapidly growing use of Javascript on hand-held mobile devices, performance ranking of web browsers among other uses [41, 42], it becomes important that we begin to start understanding what are the architectural execution characteristics of these benchmarks. Understanding these benchmarks and identifying redundancy in these benchmarks, the central theme of this paper, constitute the first step towards fair browser performance ranking and benchmarking. That is why we believe that findings in this paper are important for system software performance analysis and benchmarking communities.

In Section 2, we provide the details of experimental methodology and statistical data analysis techniques. In Section 3, we discuss the architectural execution characteristics and similarity analysis of Javascript benchmarks. Section 4 discusses related work, and finally, Section 5 concludes the paper.

2. EXPERIMENTAL METHODOLOGY

This section describes our experimental methodology for characterizing Javascript benchmarks. First, we describe hardware performance counters and microarchitecture independent characteristics used in this study. Then, we discuss statistical data analysis techniques. This is followed by the description of our benchmark suites and experimental system details.

2.1 Hardware Performance Counter Based Characterization

Hardware performance counters serve as a first-hand tool to understand workload characteristics, and hence, have been widely used in the past for architectural characterization of different workloads [9, 13, 30, 37]. The primary reason for the wide use of hardware performance counters has been its low overhead and ease of use. However, some studies have shown that considering a small set of hardware counters may lead to misleading conclusions about behavior of different programs, as they may fail to fully capture the inherent program characteristics [15]. One solution to the problem is to use micro-architecture independent characteristics to understand different program behaviors. Therefore, in this paper we use micro-architecture independent characterization methodology to understand the similarity across different programs.

However, in conjunction with microarchitecture independent characteristics, we also make use of a wide variety of available hardware performance counters, that go beyond usual characteristics such as IPC, branch misprediction rate, and cache miss rate. For example, we include amount of speculative instructions, fetch and resource stall, different types of branches, their misprediction rate, instructions per branch, branch speculative factor, modified cache lines allocated in L1 cache, L1 and L2 accesses per instruction, and L2 and memory bandwidth. We couple our discussion on hardware counter characterization with microarchitecture independent characterization to understand the difference between these two approaches, sources of such differences, and their implications.

2.2 Microarchitecture Independent Characterization

Microarchitecture independent characteristics enable us to compare the similarity between different programs based on their inherent program nature. In this study, we use a variety of microarchitecture independent characteristics that have been shown to be sufficient for statistically meaningful comparison between programs [15]. These characteristics include instruction mix, control-flow behavior, inherent instruction level parallelism, and data locality.

2.2.1 Instruction Mix

The instruction mix of a program measures the relative frequency of different types of instructions: control-flow, memory accesses (load and store), arithmetic and floating point operations, string and other operations.

2.2.2 Instruction Level Parallelism

We use dependency distance as a micro-architecture independent measure of inherent ILP in a program. Dependency distance is measured as the number of instructions between the production and consumption of a register instance. We spread dependency distance distribution across seven bins: dependency distance of one instruction, up to 2, 4, 8, 16, 32 instructions and more than 32 instructions. An inherently high ILP program is expected to show a higher relative frequency of large dependency distances.

2.2.3 Control-flow Behavior

We measure branch predictability in microarchitecture independent fashion, by using the partial match (PPM) predictor proposed by Chen et al. [8]. PPM serves as a theoretical foundation for branch predictability instead of variation of any hardware predictor implementation. We use PPM in 4 different configurations (global/local branch history, shared/separate prediction tables), using three different history length (4,8,12 bits). In addition, the average branch taken and transition count are also measured. This control-flow

characterization is similar to previous micro- architecture independent branch behavior characterization studies [15].

2.2.4 Working Set Size and Data Locality

We characterize the working set size for both the instruction and data stream by counting the number of unique 32 bytes blocks and 4KB pages touched. Data locality is characterized by using global and local data stream strides. We use the same definition of global and local data streams as used by Lau et al. [21]. We characterize read and write data streams separately unlike previous studies [27, 28], because we found that combined characterization loses important characteristics specific to each stream.

2.3 Statistical Data Analysis using PCA and Clustering Algorithms

Characterizing workloads can be a daunting task given different types of workloads, performance metrics, and associated huge characterization data. Any desired approach should analyze the data aiming two key outcomes: 1) (dis)similarity among given programs, 2) key performance limiting factors. Unfortunately, interpreting huge characterization data and drawing statistically meaningful conclusions is difficult. For example, by looking at different execution metrics such as branch misprediction rate, cache miss rate, and IPC of different programs one cannot conclude which Javascript benchmarks are similar, and what the primary bottlenecks for these Javascript benchmarks are. Furthermore, different execution metrics may interact with each other in a complex manner, prohibiting them to have simple linear correlation with IPC, and hence difficult to analyze. *To overcome this challenge, in this paper we apply principal component analysis (PCA) to analyze data obtained from both hardware counters and microarchitecture independent characterization.*

PCA applies an orthogonal transformation on a group of possibly correlated variables to convert them into a group of uncorrelated variables (principal components). Therefore, this technique is useful in reducing the dimensionality of data while retaining most of the original information. To remove the skew due to each parameter having a different range, we first normalize the data to unit normal distribution. Unit normal distribution is a distribution with zero mean and unit standard deviation. PCA analysis produces a set of uncorrelated p variables $PC_1, PC_2, \cdots PC_p$, called principal components that form a linear combination of original variables $V_1, V_2, \cdots V_p$. These principal components have decreasing variances, i.e. PC_1 having the most variance and PC_p the least. This property of PCA analysis can be used as an advantage to reduce the dimension of data without losing much of the information contained in the original data set. We may select only a few top principal components that have higher variances using standard selection criteria and control the amount of information lost.

PCA removes correlation among variables, and hence it can be used to visualize similarity between programs. In addition to PCA, we apply clustering analysis to find a subset of Javascript benchmark suite. We apply agglomerative hierarchical clustering. In hierarchical clustering, we represent a program as a vector of different execution characteristics and scale the characteristic vector as we do in PCA. Then, we calculate the euclidean distance between the vectors of different programs and apply ward method to produce hierarchical clustering that is represented as a dendrogram or tree. We use this representation to draw insights about the similarity between different Javascript benchmarks.

PCA analysis helps us in finding (dis)similarity and hence, exposes the possible redundancy in the benchmark suite, if any. This is important because having hidden redundancy in the benchmark suite skews the average towards redundant characteristics Therefore, we need to be aware of such redundancy, and make sure benchmark suite is well balanced and representative of the real world applications.

We also note that we perform PCA analysis for different characteristics individually before performing a combined PCA analysis for all characteristics together. This is because performing only one combined PCA for all characteristics together may hide some insights about benchmark characteristics. Reduced dimensions generated from PCA analysis may be linear combination of different characteristics (e.g. branch, cache, ILP, bandwidth, etc.) and one particular characteristic may appear in multiple dimensions with different weight factors associated with it. Consequently, one cannot infer the similarity across benchmarks for a particular characteristic (e.g. branch behavior) from single integrated PCA analysis. One must perform PCA analysis using different branch characteristics to infer the similarity across benchmarks for branch behavior. Stated otherwise, to quantify similarity across benchmarks based on ILP behavior, we have to perform PCA analysis using ILP characteristics, because an integrated or combined PCA analysis cannot provide such specific information.

2.4 Benchmark, Tools and Experimental System

We use two of the most widely used Javascript benchmark suites: Sunspider and V8 [2, 3]. Both Sunspider and V8 have been widely used by researchers and performance practitioners [6, 12, 24, 41, 42]. We use the latest version of both benchmark suites. Sunspider has 26 different benchmark programs and V8 consists of 7 different benchmark programs. They cover various programs such as encryption/ decryption and constraint solvers. The goal of this paper to evaluate merit, applicability, or to characterize language constructs of these programs [6, 12, 24, 34].

We use Google's V8 javascript engine for executing these benchmarks on a 2.4 Ghz Intel Core 2 Quad machine which runs linux kernel version 2.6.18. The micro-architecture comprises 32KB L1 instruction and data cache (private per core), 4MB L2 cache (shared by two cores). We used Google's V8 javascript engine as it is being used widely, including Chrome browser, due to its better or similar performance than other competitive javascript engines [43]. Figure 1(a) shows the execution time breakdown of Javascript benchmarks using Google's V8 engine (collected using Google's V8 profiler). We point out that the third component in the figure (i.e. V8 engine) also includes execution time from other shared libraries also (e.g. glibc). We notice that benchmarks spend varying fractions of execution time in javascript code, garbage collection and shared libraries (including V8 engine loaded as shared library). This information is useful for performance tuners looking to speed up a particular benchmark using Google's V8 engine as they can find the potential bottleneck phase. We also note that our hardware performance counter account for all three components (i.e. the whole execution time).

We use likwid-perfctr tool to collect hardware performance counters [5]. In the experiments, only the Javascript benchmark runs in addition to regular OS daemons. We use MICA tool for measuring microarchitecture independent characteristics [17]. Using MICA, we could not run memory reuse distance calculation tool with our infrastructure, and hence do not include that in our results. We measure only one microarchitecture independent characteristic at a time to minimize the side effects of the instrumentation code. We also faithfully try to avoid other experimental biases discussed in [25], e.g. same environment size, etc.

For comparing Javascript benchmarks to SPEC benchmarks, we

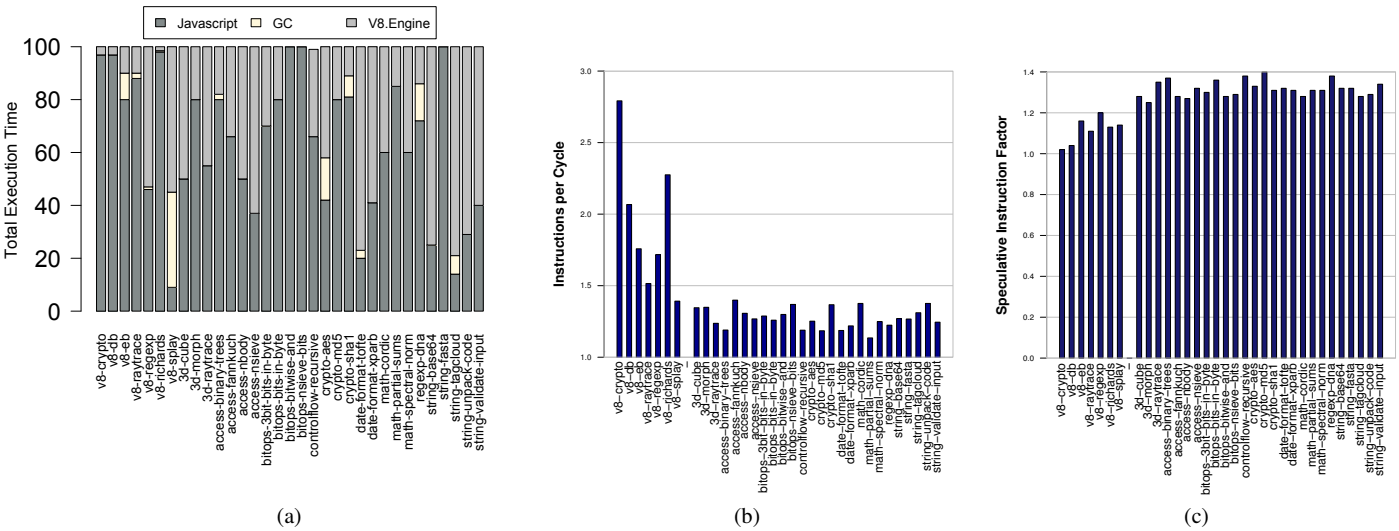


Figure 1: The execution time breakdown (a), IPC (b), and speculative inst factor (c) for V8 and Sunspider Benchmarks.

selected seven SPEC INT 2006 benchmarks: mcf, perlbench, bzip2, astar, hmmer, libquantum, Xalancbmk. The reason for not choosing all the benchmarks is the redundancy among the SPEC benchmarks [28]. Phansalkar et al. [28] showed that a subset of four programs are sufficient to represent SPEC INT benchmarks, and we point out that our seven programs constitute a super set of those four programs [28]. Therefore, these benchmarks are statistically sufficient for meaningful comparison. Adding more benchmarks make the PCA maps more crowded, and hence difficult to visualize the similarity across benchmarks (33 Javascript benchmarks and 7 SPEC benchmarks).

3. CHARACTERIZATION AND DISCUSSION

Figure 1(b) shows the IPC for both V8 and Sunspider Javascript benchmark suite on Intel Core 2 Quad machine. The V8 benchmarks are on the left, starting with prefix "v8". Sunspider benchmarks follow after the delimiter. This figure indicates that all V8 benchmarks have higher IPCs than Sunspider benchmarks. The average IPC of V8 benchmarks is 1.93, while that of Sunspider benchmarks is 1.27 with maximum being only 1.39. These varied IPCs motivate us to investigate the following architectural characteristics: 1) speculative instruction factor 2) branch characteristics 3) inherent instruction level Parallelism 4) memory access behavior.

3.1 Speculative/Wasteful Instruction Factor

Speculative instruction factor is defined as the number of decoded instructions divided by the number of retired instructions [23]. Speculative instructions can also be interpreted as wasteful instructions since they are decoded but not retired, and hence, wasting some pipeline resources. To be consistent with previous literature [23, 7], we will use the term speculative instruction factor in the rest of the paper. Figure 1 (c) shows speculative instruction factor for both benchmark suites. This figure indicates that V8 benchmarks have lower speculative instruction factor (average 1.11) than Sunspider benchmarks (average 1.31). This is consistent with our earlier observation that V8 benchmarks have higher IPCs because a higher IPC implies less wasteful work and hence, low speculative instruction factor. For example, v8-crypto has a low speculative instruction factor (1.02) corresponding to a high IPC (2.7). However, we note two exceptions: v8-raytrace and v8-splay. They have relatively lower IPCs despite low speculative instruction factor, which indicates that useless speculative work is not primarily responsible for their low IPCs. The underlying reason is other factors (data locality and resource stalls) as discussed later. The following summarizes our finding.

Finding 1. *V8 Javascript benchmarks consistently have higher IPC and lower speculative instruction factor than Sunspider benchmarks, indicating relatively less wasteful work.*

This observation hints that V8 Javascript benchmarks have high instruction level parallelism, and hence, future microprocessor designs should continue to exploit this. However, higher speculative instruction factors in Sunspider benchmarks imply decoding/executing more wasteful instructions. That may be caused by branch mispredictions. This leads us to discuss the branch characteristics next.

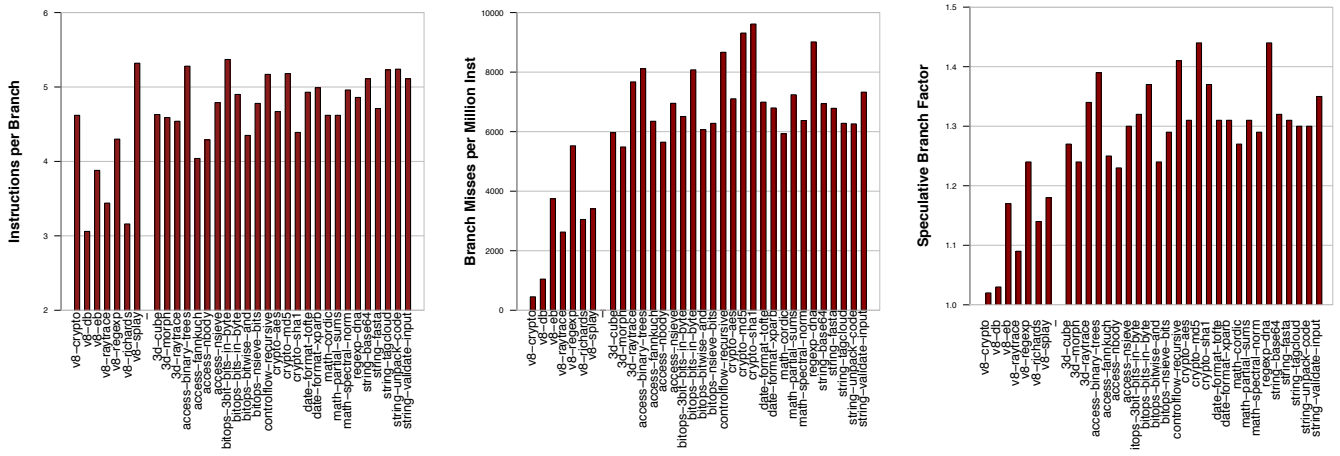
3.2 Branch Behavior Characteristics

Figure 2(a) shows the number of instructions encountered per branch, i.e. a lower value corresponds to more branches. Interestingly, V8 benchmarks have lower number of instructions per branch compared to Sunspider benchmarks, indicating more frequent occurrence of branches in V8 benchmarks instead of Sunspider benchmarks.

Sunspider benchmarks encounter branches less frequently than V8 benchmarks (Figure 2(a)), but Sunspider benchmarks still have significantly higher speculative instructions (Figure 1(c)). This behavior could be because of higher branch mispredictions, resulting in more pipeline squashes and hence, more wasteful work. This motivates us to investigate the total number of branch mispredictions.

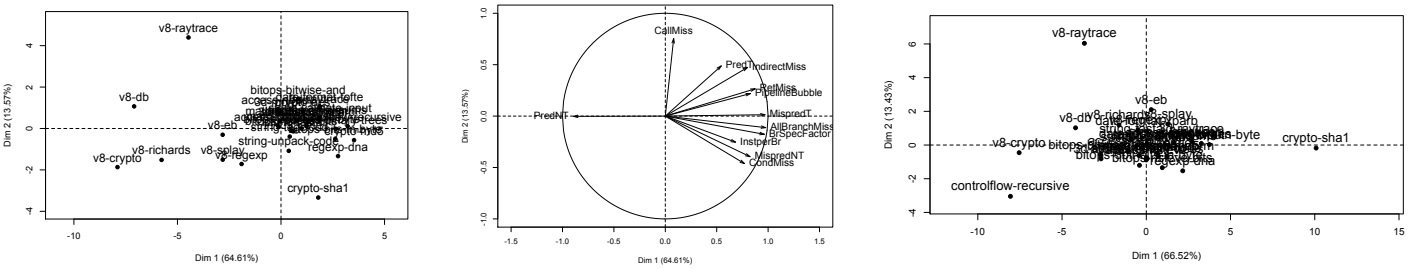
Figure 2(b) shows the number of branch mispredictions per million instructions. We notice that Sunspider benchmarks have significantly higher branch mispredictions than V8 benchmarks. This explains why Sunspider benchmarks have higher speculative instruction factor compared to V8 benchmarks. Higher branch mispredictions results in more wasteful work, and hence, higher speculative instruction factors for Sunspider benchmarks despite the fact that these benchmarks encounter branch instructions less frequently.

Figure 2(b) indicates that v8-eb and v8-regexp have the highest branch mispredictions among all V8 benchmarks, this explains why v8-eb and v8-regexp have the highest speculative instruction factor among all V8 benchmarks (Figure 1(c)).



(a) (b) (c)

Figure 2: Branch Occurrence Frequency per Inst (a), Branch Mispredictions per million Inst (b), Speculative Branch Factor (c) for V8 and Sunspider Benchmarks



(a) (b) (c)

Figure 3: Hardware counter based Branch Behavior PCA Analysis Map (a), Principal Components of Hardware-counter based Branch Behavior PCA Analysis (b), Micro-architecture independent Branch Behavior PCA Analysis Map (c).

To support these observations, we plot the speculative branch factor, i.e. the number of all decoded branches divided by all retired branches (Figure 2(c)). This figure indicates that Sunspider benchmarks have significantly higher branch speculative factor compared to V8 benchmarks, indicating higher fractions of branch squashes due to mispredictions. This is in accordance with our previous observations. The following summarizes our finding:

Finding 2. *The branch occurrence frequencies of V8 benchmarks are higher compared to those of Sunspider benchmarks. However, V8 Javascript benchmarks have significantly lower branch mispredictions than Sunspider benchmarks, resulting in less branch misprediction penalty and hence, higher IPCs.*

Branch misprediction rate and branch frequency are good metrics to gain first hand insights about branch characteristics of these benchmarks. However, these two metrics alone are not statistically sufficient to characterize (dis)similarity among these Javascript benchmarks. Do we really need all 33 benchmarks to represent branch characteristics of all Javascript programs? To investigate this, we perform PCA analysis using additional hardware performance counters for different branch characteristics, e.g. different types of branch misses (conditional, call, return, indirect), predicted but not taken, predicted taken, mispredicted but taken, instructions per branch, branch speculative factor, pipeline bubbles created due to branch mispredictions, etc.

Figure 3(a) shows the PCA map of benchmarks on two principal components that result from our PCA analysis. The key to read a PCA map is that if two benchmarks are similar they appear closer in the PCA map. Benchmarks that are not similar are farther away in the PCA map. There are several interesting findings from this analysis. First, most of the Sunspider benchmarks are clustered together though they have significant variation among them for traditional branch metrics such as branch mispredictions, and instructions per branch (Fig 2(a) and (b)). Second, V8 benchmarks are scattered on the PCA plot unlike Sunspider benchmarks. Therefore, including additional variety of branch characteristics enables us to visualize (dis)similarity across benchmarks which is otherwise non-trivial and error-prone to judge using only traditional metrics such as branch mispredictions and instructions per branch.

Unfortunately, because of the inherent black-box nature of PCA analysis, one cannot gain intuition about what factors make benchmarks dissimilar on the PCA map. For example, we know that crypto-sha1 is uniquely placed in the bottom right corner of the PCA map (Figure 3(a)), but we cannot explain what distinguishes crypto-sha1 from other benchmarks. Therefore, to gain deeper insights about benchmark dissimilarity, we plot the characteristics that constitute the principal components of PCA analysis (Figure 3(b)).

Figure 3(b) indicates that PC1 positively correlates with high branch mispredictions but negatively correlates with predicted but not-taken branches. PC2 is highly correlated with call mispredictions. This knowledge enables us to reason about the placement of different benchmarks on the PCA map and what makes them dissimilar from other benchmarks. We point out that crypto-sha1 is uniquely placed at the bottom right corner, because crypto-sha1 has the most number of mispredicted not-taken branches with almost all its branch mispredictions being conditional mispredictions – making it unique among all the benchmarks. On the other hand, v8-raytrace is positioned at the left top corner because of high call branch mispredictions as indicated by high PC2 value. v8-crypto

and v8-richards are on the left bottom quarter because of frequently predicted but not taken branches. The following summarizes our finding.

Finding 3. *Based on PCA analysis for hardware counter based branch characterization, we discover that there is significant redundancy among Sunspider benchmarks, however in contrast V8 benchmarks are dissimilar to each other(Figure 3(a)). Our PCA analysis visually reveals the degree of (dis)similarity among these benchmarks and provides an intuitive base to reason about such similarity (Figure 3(b)).*

We showed that PCA analysis based on the branch characterization is useful for gaining insights about (dis)similarity of benchmarks. To take one step further, next we investigate how hardware counter based PCA map matches up with the microarchitecture independent characterization. Figure 3(c) shows the PCA map of benchmarks on two principal components that result from our PCA analysis based on the micro architecture independent branch characteristics. Care has to be taken in comparing this PCA analysis plot with hardware counter based PCA analysis plot (Figure 3(a)) as their PCs are composed of different characteristics, therefore the absolute quadrant position of benchmarks is not important, rather the relative placement is more important for gaining insights.

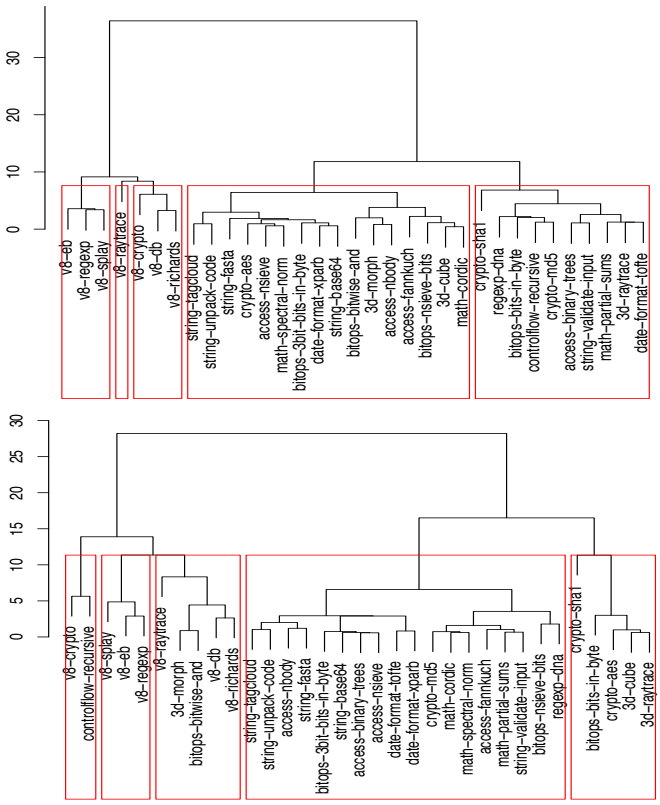


Figure 4: Dendrogram of Branch Behavior Characterization based on Hardware counters (top), Microarchitecture Independent Characteristics (bottom).

We compare these two characterization approaches to investigate if large and varied hardware counter based characteristics can match up with micro-architecture independent characterization. Interestingly, both PCA plots (Figure 3(a) and 3(c)) show similar map of benchmarks. For example, Sunspider benchmarks seemed to be grouped in one cluster in both counter based and micro-architecture

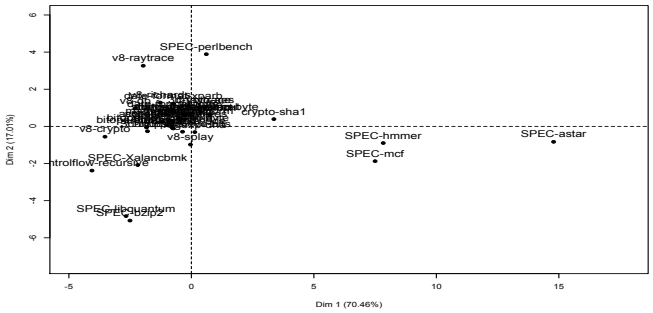


Figure 5: Microarchitecture Independent Branch Characteristics based PCA Analysis for SPEC and Javascript Benchmarks

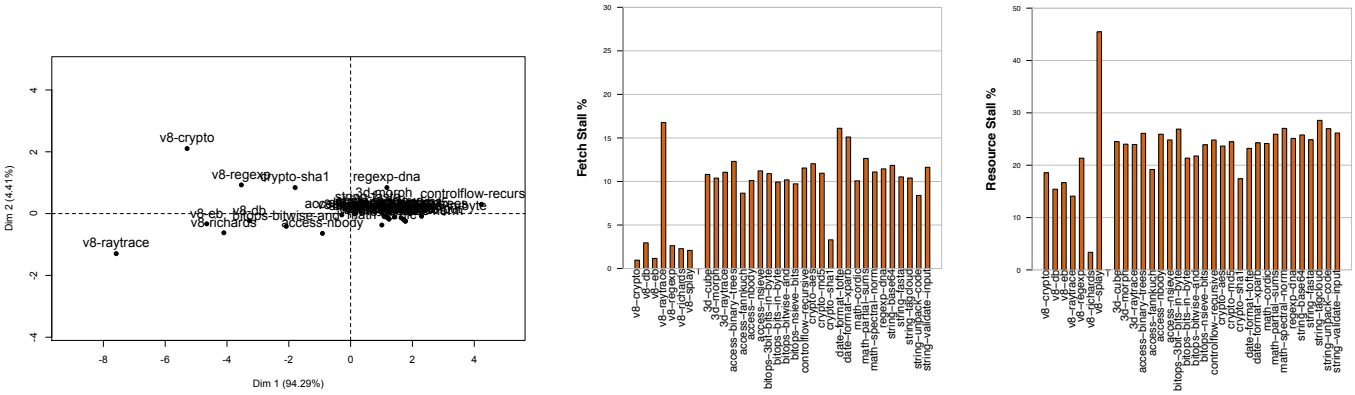
independent PCA analysis. Similarly, V8 benchmarks show significant variety in both PCA analysis maps. Hardware counter based PCA analysis (Figure 3(a)) showed that v8-raytrace and crypto-sha1 are significantly different from other benchmarks. This is confirmed by micro architecture independent PCA analysis as well (Figure 3(c)). It indicates that sufficiently large and varied hardware counter based characteristics are capable of producing useful insights about branch behavior similarity across benchmarks with less overhead compared to microarchitecture independent characterization.

This also has an implication for future processor designs. Adding a larger variety of hardware counters helps in gaining insights about benchmark characteristics more easily and quickly. Programmers and performance tuners are expected to benefit from a larger variety of hardware counters. For example, Figure 3(a) and (b) together indicate what types of branches are encountered, which types of branches are incorrectly predicted for different benchmarks, what benchmarks would benefit from improving the accuracy of return call mispredictions, etc.

However, one notable exception is Sunspider controlflow-recursive benchmark. It is an outlier in the micro-architecture independent PCA analysis but not in the hardware counter based PCA analysis.This case exposes the shortcoming of hardware-counter based approach that fails to capture the inherent dissimilarity of recursive call based program from other programs. Basically, each recursive call hinders the branch prediction mechanism because different calls may result in calculation of different sub-expression lengths. Unfortunately, a specific hardware implementation of branch predictor may not be able to completely differentiate this behavior from other inherently different branch behaviors. Following summarizes our finding:

Finding 4. *Sufficiently large and diverse hardware counters are capable of producing useful insights about branch behavior (dis)similarity across Javascript benchmarks and they closely match with microarchitecture independent characterization.*

Figure 4 shows the cluster of benchmarks as generated by the hardware counter based characterization and the micro architecture independent characterization. We note they produce similar dendrogram. In both the dendrograms, Sunspider benchmarks are divided into two major groups, and V8 benchmarks in three groups. Most of the benchmarks remain in the same group in both dendrograms, with one notable difference: controlflow-recursive is situated near to v8-crypto in micro-architecture independent dendrogram. This is because its branch behavior was not accurately captured by hardware counter based characterization as discussed earlier.



(a) (b) (c)

Figure 6: Microarchitecture independent ILP characterization (a), Fetch Stall % of Total Execution Time (b), Resource Stall % of Total Execution Time (c)

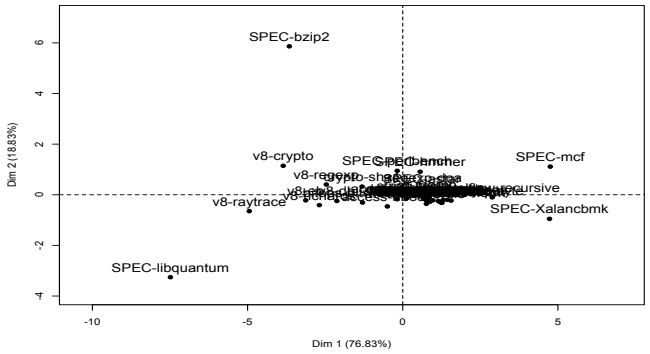


Figure 7: Microarchitecture Independent ILP Characteristics based PCA Analysis for SPEC and Javascript Benchmarks

Finally, Figure 5 shows the PCA map of Javascript benchmarks and SPEC 2006 benchmarks. SPEC benchmarks are selected based on the criteria discussed in Section 2.4. We notice that some Javascript benchmarks are significantly different from SPEC benchmarks, e.g. v8-raytrace, v8-crypto and Sunspider benchmarks. For example, v8-raytrace is unique because it suffers the most from return call mispredictions and predicted but not taken branches. Though SPEC benchmarks cover enough of space on the PCA map, adding some of the Javascript benchmarks would add more variety and hence, such an addition will be useful for benchmarking and guiding future processor designs.

3.3 Inherent Instruction Level Parallelism

Currently, on the contemporary processors, IPC is one way to quickly estimate ILP using hardware performance counters. The lack of register level dependency information hides inherent instruction level parallelism available in the program. Therefore, we employ micro-architecture independent characterization to estimate instruction level parallelism.

Using microarchitecture independent register dependency information (Section 2.2.2), we perform PCA analysis. Figure 6(a) shows the resulting PCA map. We point out that the first principal component, PC1, strongly correlates with low ILP. Therefore, a negative PC1 value indicates a high ILP. Hardware counter based counterpart analysis has only one available characteristic (IPC), and that is plotted in Figure 1(b).

First, we observe that similar to the IPC graph (Figure 1)(b), PCA analysis map (Figure 6(a)) shows that Sunspider benchmarks are clustered in a low IPC area, with few exceptions crypto-sha1, bitops-bitwise-and, and access-nbody benchmarks that have high inherent ILP. Second, this PCA map indicates V8 benchmarks have inherently high, though varying amount of, ILP that also conforms with our IPC graph. This indicates that even some of the Sunspider benchmarks have high ILP that future processor may aim to exploit.

The most interesting insight is that the hardware counter based characterization (IPC plot Figure 1(b)) cannot correctly capture the order of V8 benchmarks according to available inherent ILP in the program, something that is captured by micro-architecture independent PCA map (Figure 6(a)). For example, v8-raytrace has the highest ILP according to PCA map but its IPC as reported by the hardware counter based analysis is not the highest (Figure 1). Hardware counter based analysis can not correctly capture the order of ILP because of several complex microarchitectural interaction among different microarchitecture resources (iTLB, dTLB, instruction cache etc.). Microarchitecture independent analysis, on the other hand, does not suffer from the interaction problem. However, microarchitecture independent analysis lacks the ability to pinpoint the bottleneck, in contrast to hardware counter based analysis, as we show next.

To investigate the reasons for the low IPC in v8-raytrace, despite hidden inherent ILP we plot the fetch stall time as a fraction of total execution time in Figure 6(b) for all Javascript benchmarks. We notice that v8-raytrace suffers the most among all V8 benchmarks due to fetch stall. High instruction cache, iTLB and dTLB misses (shown later in Figure 8(a) and (b)) confirm this behavior. This illustrates that microarchitecture independent analysis points out the maximum realizable performance limits (ILP) in microarchitecture independent fashion. Using that information, hardware counter based characterization can pinpoint the source of performance bottleneck (fetch stall).

In another interesting case, we note that v8-eb and v8-richards are neighbors on the PCA plot, although they have significantly different IPCs (Figure 1) and yet similar Fetch stalls, iTLB, and dTLB misses (Figure 6(b), 8(a) and (b)). To investigate this, we plot resource stall time as a fraction of the total execution time in Figure 6(c). v8-eb has significantly higher resource stall compared to v8-richards, explaining the difference in IPC. Again hardware counter based characterization is able to pin-point the source of performance bottleneck when used in conjunction with microarchitecture independent analysis.

These cases indicate that while hardware counter based characteristics may not capture the ILP potential correctly, they do have the potential to break down the performance limiting factors on a given architecture which micro-architecture independent analysis may not provide. Therefore, it makes a better case to use microarchitecture independent and hardware counter based analysis jointly

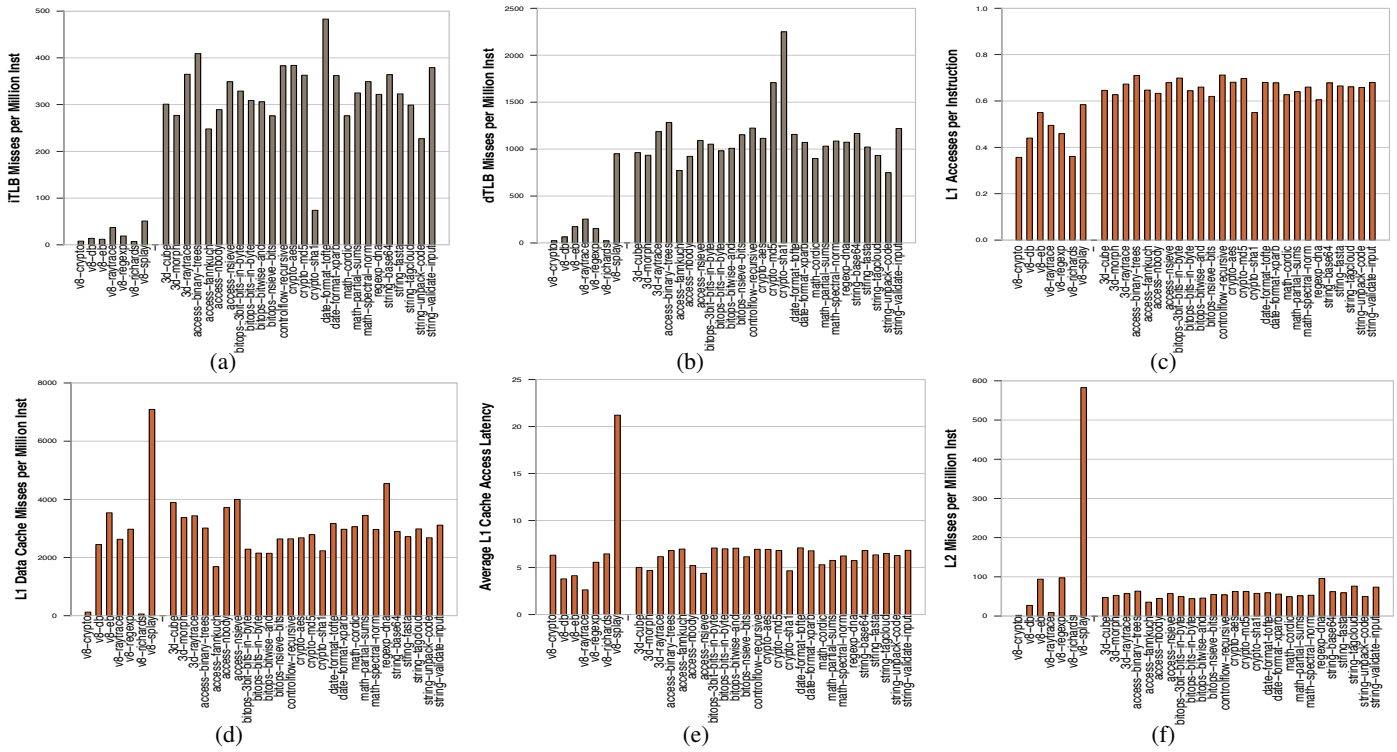


Figure 8: Instruction TLB misses per million Inst (a), data TLB misses per million Inst (b), L1 accesses per Inst (c), L1 misses per million Inst (d), Average L1 access latency (e), L2 misses per million Inst (f) for V8 and Sunspider Javascript Benchmarks

to pinpoint performance bottlenecks on a given architecture.

We note that hardware counter based characterization does not match up with microarchitecture independent characterization because of the lack of variety of ILP related hardware counters. This implies that there is a need for more versatile hardware counters in the future processor designs indicating the degree of inherent ILP in the program, for example the percentage of instructions stalls due to RAW, WAR, and WAW dependency, register dependency distribution etc.. This will give an idea to programmers and performance tuners about how much ILP is inherently available in the program. Following summarizes our finding:

Finding 5. *Microarchitecture independent analysis reveals the hidden ILP in programs. Using this information, hardware counter based analysis can pinpoint the sources of a performance bottleneck (resource constraint, fetch bottleneck etc.) which inhibits the full realization of the ILP. Also, we showed why there is a need for more versatile hardware counters in future processors that capture the degree of inherent ILP in the program.*

Next, we discuss a special case of v8-splay which shows a low IPC (Figure 1). Hence, a high resource stall fraction (as shown in Figure 6(c)) may lead one to conclude that v8-splay may have a potentially high IPC but is severely limited by resource constraints and excessive L1 data cache misses (as shown later in Fig 8(d)). However, this is not the case. Microarchitecture independent analysis indicates (Figure 6(a)) that inherent ILP in v8-splay is limited, and is not hidden due to resource or other constraints.

We note that while hardware counter based analysis struggles to correctly isolate different behaviors (ILP and MLP), microarchitecture independent analysis is able to point out that ILP in v8-splay is already limited in architecture independent fashion and is not an artifact of high resource stall or data cache misses. Such cases emphasize why a joint analysis of hardware counters and micro-architecture independent characteristics is necessary for understanding performance behaviors correctly.

Finally, Figure 7 shows an ILP based PCA map of Javascript benchmarks and SPEC 2006 benchmarks. Similar to branch characteristics, we notice that some Javascript benchmarks are significantly different from SPEC benchmarks in terms of ILP characteristics (e.g. v8-raytrace, v8-crypto). However, we also point out that dissimilarity between these two benchmark suites is not as large as branch characteristics since some of the Sunspider benchmarks and SPEC benchmarks overlap.

3.4 Memory Access Characteristics

In this section, we first present hardware counter based characterization of memory access behavior, perform PCA analysis and then compare it with microarchitecture independent characterization to gain additional insights.

For memory access characterization, we first present characterization of TLB misses, cache misses, and then bandwidth requirements of memory and L2 cache.

Figure 8(a) and (b) show instruction and data TLB misses per million instructions respectively. We observe that Sunspider benchmarks have higher instruction and data TLB misses than V8 benchmarks, with one notable exception being v8-splay. We noticed earlier in Section 3.1 that v8-splay has lower IPC among V8 benchmarks despite a low speculative instruction factor, the reason lies in data access behavior. v8-splay has a very high data and instruction TLB misses compared to other V8 benchmarks.

Figure 8(c), (d), and (e) show L1 accesses per instruction, L1 data cache misses per million instructions, and L1 data cache average latency respectively. We notice that Sunspider benchmarks have higher data accesses per instructions compared to V8 benchmarks. However, there is a significant variation across V8 bench-

marks in data cache misses per million instructions, with v8-splay benchmark having the highest L1 data cache misses among all benchmarks. The average L1 data cache access latency for Sunspider benchmarks (6.2 cycles on an average) is approximately 30% higher than V8 benchmarks (4.8 cycles on average, excluding v8-splay).

Finding 6. *Our hardware counter based characterization reveals that compared to V8 benchmarks, Sunspider benchmarks have higher TLB misses, L1 data accesses, per million instructions, and higher average L1 access latency.*

Next, we plot L2 cache misses per million instructions (Figure 8(f)). Figure 8(f) indicates that L2 cache misses per million instruction is uniformly low across all benchmarks (on average 70 misses) except v8-splay and that is supported from our other results as that particular benchmark suffers from high L1 cache misses and data TLB misses due to a large working set size (traversal of splay tree structure). Consequently, Javascript benchmarks may not gain significant benefit much from larger L2 caches in future design processors (except v8-splay). Rather they are more likely to see extra benefit from better TLB and L1 cache designs in future processors.

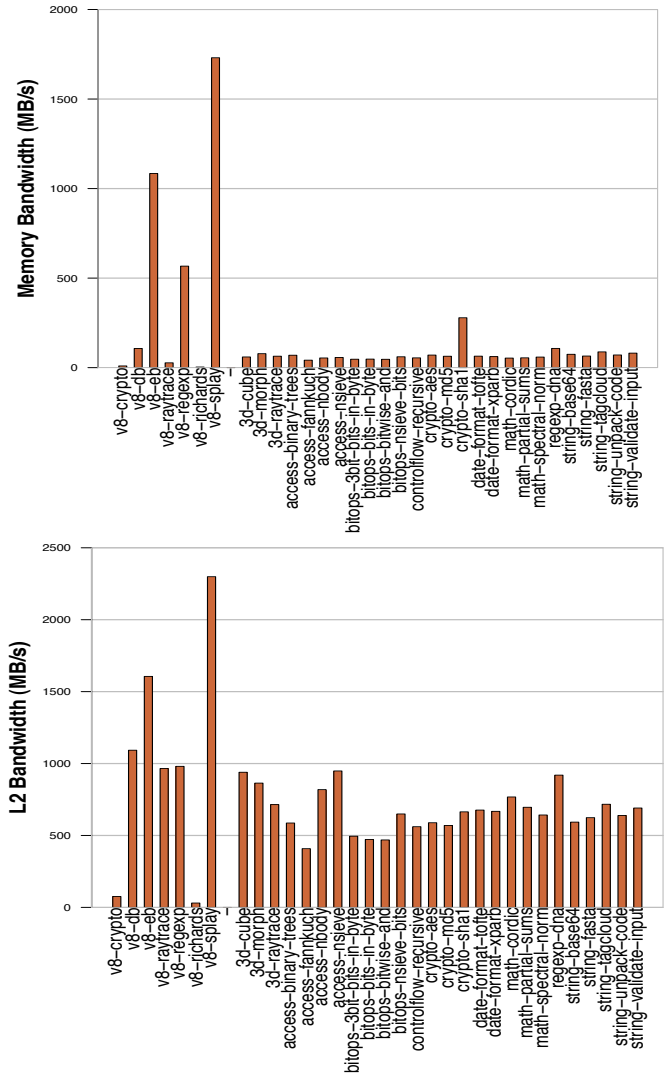


Figure 9: Memory Bandwidth in MB/sec (top), L2 cache Bandwidth in MB/sec (bottom).

Now, we look at memory bandwidth requirements of Javascript benchmarks. Figure 9 (top) shows memory bandwidth per second for all benchmarks. First, we notice Sunspider benchmarks have relatively low memory bandwidth usage (approximately 100 MB/s). On the other hand, V8 benchmarks have varying memory bandwidth usage. We can explain that by observing L2 caches misses per million instructions (Figure 8(f)). Benchmarks v8-crypto, v8-raytrace and v8-richards have less than 10 L2 cache misses and hence low memory bandwidth usage. While v8-db, v8-eb, v8-regexp, and v8-splay have high L2 cache misses and consequently a high memory bandwidth usage, they can be as high as 1.5GB/sec though we note that it is under what current contemporary processors provide. Though limited off-chip memory bandwidth is unlikely to grow fast in future, Javascript benchmarks are not likely to aggravate the problem.

Some Sunspider benchmarks have comparable L2 cache misses per million instructions to V8 benchmarks (v8-eb and v8-regexp), still their memory bandwidth usage is significantly lower than that of those V8 benchmarks. This observation is an artifact of two facts: the metric chosen for bandwidth usage (MB/s rather than MB per million instructions) and IPC. For V8 and Sunspider benchmarks which have similar L2 cache misses per million instructions, V8 benchmarks have higher IPCs than Sunspider benchmarks. This reveals that V8 benchmarks are able to generate more memory requests per second than Sunspider counterparts, resulting in a higher memory bandwidth usage.

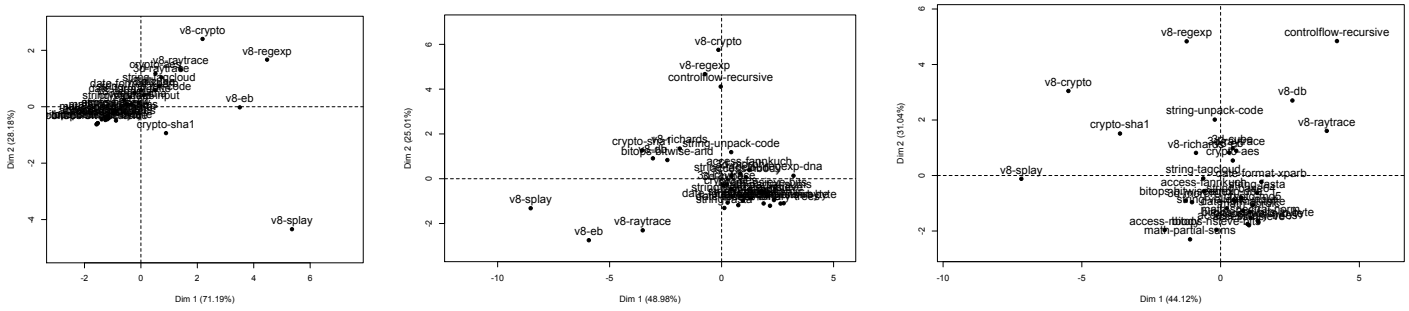
Figure 9 (bottom) shows L2 cache bandwidth per second for all benchmarks. We notice that the L2 cache bandwidth usage is high for all benchmarks and is a mirror of L1 cache misses per million instructions (Figure 8(d)). We note that Sunspider benchmarks fall short of V8 benchmarks' L2 cache bandwidth requirement despite similar L1 cache misses per million instructions, this can be explained using same high IPC reasoning as explained earlier.

Finding 7. *We find that V8 benchmarks have higher memory and L2 cache bandwidth usage compared to Sunspider benchmarks despite their comparatively lower L1 and L2 cache misses per million instructions. We explain that high IPC of V8 benchmarks is responsible for this behavior.*

Now we discuss microarchitecture independent characterization of memory and instruction access behavior. First, we plot the PCA analysis map of instruction and data footprint in Figure 10(a). We point out that PC1 corresponds to larger footprint (both data and instruction). A positive PC2 corresponds to a larger instruction footprint and a negative PC2 corresponds to large data footprint. In Figure 10(a), v8-splay and crypto-sha1 are in the bottom right corner indicating a larger data footprint. This is also confirmed by high data TLB and L1 cache misses (Figure 8(b) and (d)). We also note that these benchmarks have a low instruction TLB misses explaining why they have a negative PC2. Similar to hardware counter based analysis, most of other Sunspider benchmarks are clustered in one group.

On the other hand, v8-raytrace, v8-crypto and v8-regexp show large instruction footprints as indicated by PCA analysis. However, the hardware counter based characterization does not show such a trend, because instruction TLBs may be large enough and accesses may have good locality that it shadows the large instruction footprint. This explains the source of difference between these two characterization approaches.

Next, we study data stream strides patterns. Previous studies [27, 28] have performed PCA analysis on stream strides without differentiating between type of access (read or write). In this study, we perform PCA analysis separately for read and write data streams.



(a) (b) (c)

Figure 10: PCA analysis map for Data and Instruction footprint (a), Data read locality (b), Data write locality (c) using Micro architecture independent characterization.

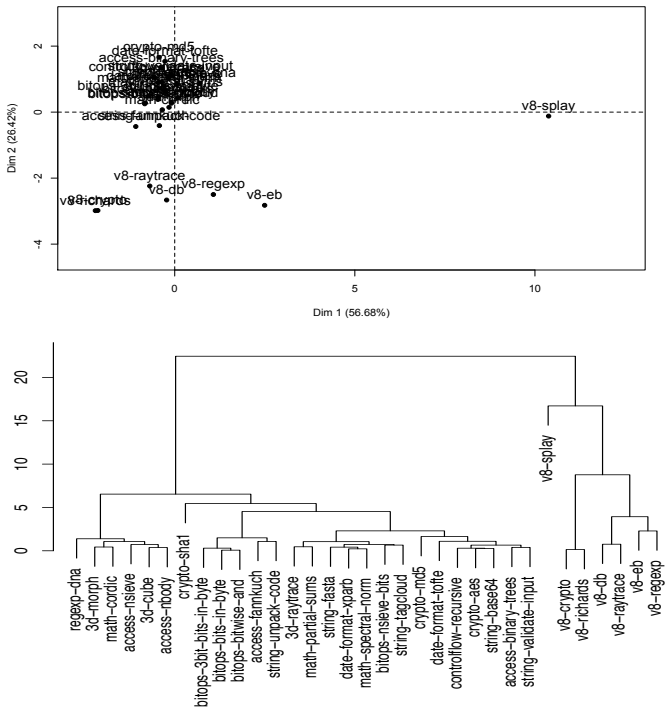


Figure 11: PCA analysis map using hardware counter based memory characteristics (top), and its corresponding cluster of benchmarks (bottom).

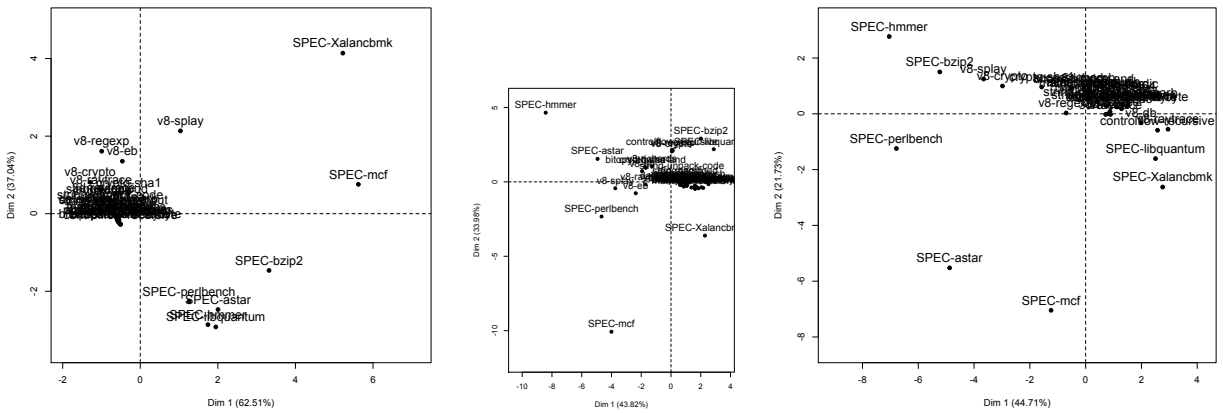
Figure 10(b) presents map of benchmarks generated by PCA analysis on read stream stride. We can make a couple of observations. First, most of the benchmarks are clustered together near the origin. Second, there is a group of benchmarks at the top, consisting v8-crypto, v8-regexp, controlflow- recursive, that have very short read strides and only one dominant stride. Finally, in the bottom left corner three benchmarks (v8-splay, v8-db and v8-raytrace) show up to three dominant strides. This implies that stride prefetching for Javascript benchmarks should work well.

Figure 10(c) presents a map of benchmarks generated by PCA analysis on write stream stride. First, we note that the map is not similar to read stream stride PCA map indicating that write strides are indeed quite different. In Figure 10(c), one distinctive cluster of Sunspider benchmark at the bottom indicates benchmarks with two to three dominant write strides, while other benchmarks show behavior that cannot be clubbed together because of the difference in the number of dominant strides and their lengths.

Now, we investigate similarity across benchmarks based on TLB, cache and bandwidth characteristics. Figure 11 (top) shows a PCA map using hardware counter based characterization, and Figure 11 (bottom) shows the resulting cluster of benchmarks. We make a few important observations. First, the PCA map indicates that V8 benchmarks show a significant variety in instruction and memory access characteristics, while Sunspider benchmarks are largely clustered at one point. This is also confirmed by the cluster of benchmarks. Second, v8-splay is the most unique benchmark as indicated by the high linkage distance in Figure 11 (bottom). This is primarily because its large data footprint. We observer similar results for microarchitecture independent PCA and cluster analysis (not presented here for brevity).

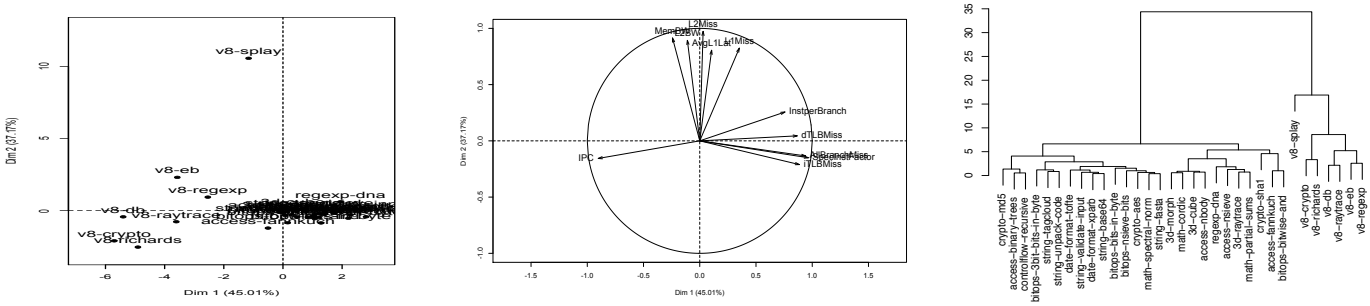
Next, we perform micro-architecture independent characterization of memory access behavior for both SPEC and Javascript benchmarks. The PCA analysis map of instruction and data footprint is shown in Figure 12(a), read stride stream PCA in Figure 12(b), and write stride stream PCA in Figure 12(c). We notice that there is significant overlap between SPEC and Javascript benchmarks in terms of read and write strides, in fact SPEC benchmarks have much more variation than Javascript benchmarks. However, instruction and data footprint PCA analysis shows that Javascript benchmarks are significantly different than SPEC benchmarks, e.g. v8-splay, v8-regexp, and v8-eb, etc. Therefore, similar to other characteristics, this indicates that adding some of these V8 Javascript benchmarks would add more variety and hence, such an addition is expected to be useful for guiding future processor design.

Finally, we present three additional plots. Figure 13(a) shows PCA map of all benchmarks using all hardware counter based characteristics, Figure 13(b) shows principal components of this PCA analysis, and finally Figure 13(c) shows cluster of benchmarks generated using this PCA analysis. We can make a couple of interesting observations. First, V8 benchmarks have a significant variety considering all execution characteristics we measured, in contrast to Sunspider benchmarks. Second, the PCA map quantifies the degree of (dis)similarity across benchmarks. Third, the cluster dendrogram enables us to choose as many unique benchmarks as we like by drawing a horizontal line at different linkage distances along the y-axis. Clearly, 4 out of 7 V8 benchmarks are unique, while Sunspider benchmarks may only have 5 to 7. Third, Figure 13(b) shows relative importance of different components (we only show most important ones for clarity). It indicates that high IPC benchmarks will be on the left of PCA map as confirmed by the placing of V8 benchmarks. Our PCA analysis(Figure 13(b)) shows that high number of branches, branch mispredictions, TLB misses contribute more towards low IPC. In comparison, cache misses, and bandwidth requirements have relatively smaller impact on IPC for Javascript benchmarks. This has implications for future processor



(a) (b) (c)

Figure 12: PCA analysis map of both SPEC and Javascript Benchmarks for Data and Instruction footprint (a), Data read locality (b), Data write locality (c) using Micro architecture independent characterization .



(a) (b) (c)

Figure 13: PCA analysis map using all hardware counter based execution characteristics (a), main principal components of the same PCA (b), and Overall clustering of benchmarks (c).

design looking to achieve better performance on Javascript workloads. They should primarily focus on reducing branch mispredictions, better L1 cache and TLB design, while L2 cache design and memory bandwidth are secondary targets. The following summarize our findings in this section.

Finding 8. *Javascript benchmarks are more performance sensitive to the number of branches, branch mispredictions, and TLB misses, in comparison to cache misses, and bandwidth usage. Therefore, future processor designs are likely to see larger performance benefit for Javascript benchmarks if they improve branch predictor, iTLB and dTLB design. Also, we show that adding some of these V8 Javascript benchmarks to more generic benchmark suites, e.g. SPEC, would add more variety and hence, such addition is expected to be useful for guiding future processor designs.*

4. RELATED WORK

Recently, there have been several efforts trying to improve performance of V8 and Sunspider Javascript benchmarks. Fortuna et al. conducted a limited study on the potential parallelism in Javascript benchmarks, and reported an average potential speed up of 8.9x [12]. Anderson et al. proposed an architectural support for Javascript type-checking on mobile processors [6]. Mehrara et al. developed an runtime system for identifying parallelizable sections in Javascript code, execute that region on multi-core platform in parallel, thereby improve the performance of Javascript codes [24]. However, none of these studies have investigated architectural execution characteristics of these Javascript benchmarks – the central theme of this paper.

There is a large body of work characterizing various workloads using hardware performance counters [40, 33, 23, 30, 31, 32, 37, 13, 9, 35, 26, 10, 11, 38, 36, 39, 19, 29]. Hoste and Eeckhout are credited for studying microarchitecture independent workload characterization, which they applied to study different workloads [15, 17]. Hoste et al. used inherent program similarity for performance prediction [16]. Using micro-architecture independent analysis, Phansalkar et al. studied similarity across SPEC CPU benchmarks [27]. They also carried out similarity analysis on SPEC 2006 benchmarks using hardware performance counter approach [28], among other related studies [14, 16, 20]. Barr reduced architectural simulation time using micro-architecture independent snapshots [1]. However, to the best of our knowledge there is no prior work studying architectural execution characteristics of Javascript benchmarks, or applying statistical techniques to discover redundancy in Javascript benchmarks – one of the key contributions of this work.

5. CONCLUSION

Due to the growing popularity of Javascript, it is important to understand the architectural execution characteristics of Javascript benchmarks, given performance deficiencies of Javascript codes due to bytecode interpretation. In this paper, we presented an in-depth architectural characterization of the widely used V8 and Sunspider Javascript benchmarks. We find that V8 benchmarks have a high instruction level parallelism despite the high branch frequencies. On the other hand, Sunspider benchmarks comparatively suffer more from a high branch misprediction penalty, high TLB and L1 cache misses, limiting their IPCs. Our analysis reveals that there is significant redundancy among Javascript benchmarks. We believe this is a critical finding as these Javascript benchmarks are being widely used for important uses such as performance ranking

of web-browsers'. Moreover, we also showed how statistical data analysis enables us to do more meaningful characterization analysis, reveals correlation among execution metrics, and explains their implications on overall performance.

6. REFERENCES

[1] Kenneth C. Barr. Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots. *PhD Thesis, MIT*, 2006.

[2] Sunspider Javascript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[3] Google V8 Benchmarks. http://code.google.com/apis/v8/benchmarks.html.

[4] JavaScript Usage Statistics. BuiltWith. http://trends.builtwith.com/javascript.

[5] LikWid Performance Counter. http://code.google.com/p/likwid/wiki/Introduction.

[6] Anderson et al. Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors". *In the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[7] Bhandarkar et al. Performance characterization of the Pentium Pro processor. *In the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 1997.

[8] Chen et al. Analysis of branch prediction via data compression. *In the Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[9] Cho et al. Workload Characterization of Biometric Applications on Pentium 4 Microarchitecture,. *In the Proc. of International Symposium on Workload Characterization (IISWC)*, 2005.

[10] Eeckhout et al. How Java Programs Interact with Virtual Machines at the Microarchitectural Level. *In the Proc. of Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[11] Esmaeilzadeh et al. Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling. *In the Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[12] Fortuna et al. A Limit Study of JavaScript Parallelism. *In the Proc. of International Symposium on Workload Characterization (IISWC)*, 2010.

[13] Ganesan et al. A Performance Counter Based Workload Characterization on BlueGene/P. *In the Proc. of International Conference on Parallel Processing (ICPP)*, 2008.

[14] Heirman et al. Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads. *In the Proc. of International Symposium on Workload Characterization (IISWC)*, 2011.

[15] Hoste et al. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. *In the Proc. of International Symposium on Workload Characterization (IISWC)*, 2006.

[16] Hoste et al. Performance Prediction Based on Inherent Program Similarity. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006.

[17] Hoste et al. Microarchitecture-Independent Workload Characterization. *In IEEE Micro*, 2007.

[18] Hsu et al. Exploring the Cache Design Space for Large Scale CMPs. *In the Workshop on Design, Architecture, and Simulation of Chip Multi-Processors (dasCMP)*, 2005.

[19] HW Cain et al. An Architectural Evaluation of Java TPC-W. *In the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2001.

[20] Isen et al. On the Representativeness of Embedded Java Benchmarks. *In the Proc. of International Symposium on Workload Characterization (IISWC)*, 2008.

[21] Lau et al. Structures for Phase Classification. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[22] Li et al. Using Complete System Simulation to Characterize SPECjvm98 Benchmarks. *In the Proc. of International Conference on Supercomputing (ICS)*, 2000.

[23] Luo et al. Workload Characterization of Multithreaded Java Servers . *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001.

[24] Mehrara et al. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. *In the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[25] Mytkowicz et al. Producing wrong data without doing anything obviously wrong! *In the Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[26] Ould-Ahmed-Vall et al. Characterization of SPEC CPU2006 and SPEC OMP2001: Regression Models and their Transferability. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[27] Phansalkar et al. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[28] Phansalkar et al. Analysis of Redundency and Application Balance in the SPEC CPU2006 Benchmark Suite. *In the Proc. of International Symposium on Computer Architecture (ISCA)*, 2007.

[29] R Hankins et al. Scaling and characterizing database workloads: Bridging the gap between research and practice. *In the Proc. of International Symposium on Microarchitecture (MICRO)*, 2003.

[30] Radhakrishnan et al. Execution Characteristics of Object Oriented Programs on the UltraSPARC-II. *In the Proc. of International Conference on High Performance Computing (HiPC)*, 1998.

[31] Radhakrishnan et al. A Performance Study of Modern Web Applications. *In the Proc. of Euro-Par*, 1999.

[32] Radhakrishnan et al. Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels. *In the Proc. of International Conference on Computer Design (ICCD)*, 1999.

[33] Radhakrishnan et al. Architectural Issues in Java Runtime Systems . *In the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2000.

[34] Ratanaworabhan et al. JSMeter: Measuring JavaScript Behavior in the Wild. *MSR-TR-2010-8*, 2010.

[35] Shuf et al. Characterizing a Complex J2EE Workload: A Comprehensive Analysis and Opportunities for Optimizations. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[36] Sweeney et al. Using hardware performance monitors to understand the behavior of java applications. *In Virtual Machine Research And Technology Symposium*, 2004.

[37] Talla et al. Execution Characteristics of Multimedia Applications on a Pentium II Processor . *In the Proc. of International Performance Computing and Communications Conference (IPCCC)*, 2000.

[38] Teng et al. Understanding the Cost of Thread Migration for Multi-Threaded Java Applications Running on a Multicore Platform. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[39] Wisniewski et al. Performance and environment monitoring for whole-system characterization and optimization. *In PAC2 Conference on Power/Performance Interaction with Architecture, Circuits, and Compilers*, 2004.

[40] Zaparanuks et al. Characterizing the design and performance of interactive java applications. *In the Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[41] Firefox is No 1 Again. http://tinyurl.com/3cww59q.

[42] JavaScript Performance Rundown. http://tinyurl.com/6fhofh.

[43] Google V8. http://code.google.com/apis/v8/design.html.