Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing

YUHAO ZHU, The University of Texas at Austin
VIJAY JANAPA REDDI, The University of Texas at Austin

Mobile applications are increasingly built using Web technologies as a common substrate to achieve portability and to improve developer productivity. Unfortunately, Web applications often incur large performance overhead, directly affecting user quality-of-service (QoS) experience. Traditional techniques in improving mobile processor performance has mostly been adopting desktop-like design techniques such as increasing single-core microarchitecture complexity and aggressively integrating more cores. However, such a desktop-oriented strategy is likely coming to an end due to the stringent energy and thermal constraints that mobile devices impose. Therefore, we must pivot away from traditional mobile processor design techniques in order to provide sustainable performance improvement while maintaining energy efficiency.

In this paper, we propose to combine hardware customization and specialization techniques to improve the performance and energy efficiency of mobile Web applications. We first perform design-space exploration (DSE) and identify opportunities in customizing existing general-purpose mobile processors, i.e., tuning microarchitecture parameters. The through DSE also lets us discover sources of energy inefficiency in customized general-purpose architectures. To mitigate these inefficiencies, we propose, synthesize, and evaluate two new domain-specific specializations, called the Style Resolution Unit and the Browser Engine Cache. Our optimizations boost performance and energy efficiency at the same time while maintaining general-purpose programmability. As emerging mobile workloads increasingly rely more on Web technologies, the type of optimizations we propose will become important in the future and are likely to have long-lasting and widespread impact.

CCS Concepts: •**Computer systems organization** →Special purpose systems;

Additional Key Words and Phrases: Web browsing, accelerator, specialization, customization, software-managed cache

ACM Reference format:
Yuhao Zhu and Vijay Janapa Reddi. 2010. Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing. *ACM Trans. Comput. Syst.* 9, 4, Article 39 (March 2010), 30 pages.
DOI: 0000001.0000001

1 INTRODUCTION

The proliferation of mobile devices and the fast penetration of Web technologies (such as HTML, CSS, and JavaScript) has ushered in a new era of *mobile Web computing*. One key driving force is the "write-once, run-anywhere" feature of Web technologies that greatly improves developers' productivity and addresses the notorious

This work is supported by Intel Corporation, AMD Corporation, and Google. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. This article extends an earlier version that appeared in the Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14) [89].
Author's addresses: Y. Zhu and V. Janapa Reddi, Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX, USA; emails: yzhu@utexas.edu, vj@ece.utexas.edu.

DOI: 0000001.0000001

device fragmentation issue [72]. To encourage developing mobile applications using Web technologies, major mobile Operation Systems such as iOS and Android expose APIs to allow easy integration of various Web browser features into mobile applications. Such a Web-based development strategy has been widely adopted by popular mobile Apps such as Uber and Instagram. A recent study by Strategy Analytics shows that 63% of all business mobile applications are now based on Web technologies [73].

The platform portability that Web technologies bring, however, is often encumbered by large performance overhead, which directly affects user quality-of-service (QoS) experience. Mobile users react to poor QoS experience by abandoning Web services, leading to severe consequences. For example, 25% of mobile users abandon Web services that take over 4 seconds to load [44]. Google estimated that "a 400 ms delay leads to a 0.44% drop in search volume." [36]. Similarly, a 1-second delay in webpage load time costs Amazon $1.6 billion annual lost in sales [21].

Traditionally, Web application performance has been network limited. However, this trend is changing. Prior work has shown that over the past decade, network technology advancements have managed to keep webpage transmission overhead almost stable, whereas the client-side computational time has increased by as much as 10X [88]. This trend is partially caused by dramatic improvement in network latency (e.g., about 10X improvement in round-trip time from 3G to LTE [37]), and partially caused by the ever-increasing computational requirement posed by new Web technologies (e.g., CSS3 and HTML5). The combined effects of faster network performance and higher computation demand indicates that future mobile Web performance will be unattainable without improving mobile CPU performance [87].

Conventional techniques in improving mobile CPU performance has largely been simply adopting desktop-oriented design techniques both in terms of single-core and multi-core performance scaling [31]. For example, mobile CPUs have gone from in-order to out-of-order microarchitecture for single-core designs (e.g., ARM Cortex-A8 to A15), and have also gone from single core to multicore (e.g., Exynos 5410 in Samsung Galaxy S4 has eight cores). However, such a desktop-like design strategy is power-hungry, and is likely coming to its end because the performance benefits do not sufficiently make up for the additional power consumption, leading to excessive energy consumption. Recent studies suggested that the advancement in Lithium-ion battery density has slowed down significantly, and as such the energy budget for mobile devices is unlikely to drastically increase in the short term [10, 71]. Meanwhile, current mobile CPU designs have already reached the thermal constraints of mobile devices, further capping performance improvement [31]. We must deviate away from traditional mobile CPU design techniques in order to provide sustainable performance improvement without broaching against thermal and energy barriers.

We see the domain-specific specialized architecture as a promising approach for future mobile CPU designs. Domain-specific specialization has long been deemed as extremely high-performance and energy-efficient because it aggregates hundreds of operations in a few instructions, and therefore reduces major sources of inefficiencies in general-purpose CPUs [32, 51, 82]. The key challenge of applying architectural specialization to Web computing is how to *retain general-purpose programmability*. The general-purpose programmability is a particular necessity for Web technologies because they involve large pieces of software that are written in a combination of different general-purpose programming languages. For example, Google's Chrome Web browser is developed in 29 languages with over 17 million lines of code [62]. Recent work have demonstrated the importance and feasibility of balancing general-purpose programmability and specialization in various data computation domains (e.g., H.264 encoding [32], convolution [67]).

Following the same architecture design philosophy, we propose the WebCore, *a general-purpose core customized and specialized for the mobile Web applications*. In comparison to prior work that either takes a fully software approach on general-purpose processors [16, 56] or a fully hardware specialization approach [11], our design strikes a balance between the two. On one hand, WebCore retains the flexibility and programmability of a general-purpose core. It naturally fits in the multicore SoC that is already common in today's mainstream mobile

devices. On the other hand, it achieves energy-efficiency improvement via modest hardware specializations that create closely coupled datapath and data storage.

We begin by examining existing general purpose designs for the mobile Web applications. Through exhaustive design space exploration, we find that existing general purpose designs bear inherent sources of energy-inefficiency. In particular, instruction delivery and data feeding are two major bottlenecks. We show that customizing current designs by properly sizing key design parameters achieves better energy efficiency. The customization step ensures that further optimizations are performed upon an optimized general-purpose baseline.

Building on the customized general-purpose baseline, we develop specialized hardware to further overcome the instruction delivery and data feeding bottlenecks. We propose two new optimizations: the *"Style Resolution Unit"* (SRU) and a *"Software-Managed Browser Engine Cache."* The SRU is a hardware accelerator for the critical style-resolution kernel within the Web browser engine. It is based on the observation that the style-resolution kernel has abundant fine-grained parallelism that is hidden in a software implementation but can be captured by a dedicated hardware structure. SRU employs a GPU-like multi-lane architecture to exploit the inherent parallelism. Through exploiting the parallelism, the SRU aggregates enough computations in a few operations, which effectively increases the arithmetic intensity and offsets the instruction delivery and data feeding overhead.

The proposed browser engine cache structure improves data feeding efficiency by exploiting the unique data access pattern of the browser engine's principal data structures such as the DOM tree and the Render tree. Web applications typically operate on one DOM/Render tree node heavily and traverse to the next one, indicating both heavy data reuse and predictable access pattern. The browser engine cache uses a small and energy-conserving hardware memory to capture the heavy data reuse and uses software to predict the access pattern and to manage the cache. Overall, the browser cache achieves a high hit rate for the important data structures but with extremely low accessing energy.

Our results show that customizations alone on the existing general-purpose mobile processor design lead to 22.2% performance improvement and 18.6% energy saving. Our specialization techniques achieve an additional 9.2% performance improvement and 22.2% overall energy saving; the accelerated portion itself achieves up to 10X speedup. Finally, we also show that our specialization incurs negligible area overhead. More importantly, such overhead, if dedicated to tuning already existing general-purpose architectural features (e.g., caches), lead to much lower energy-efficiency improvements.

In summary, we make the following key contributions in this paper:

(1) We perform thorough design-space exploration of mobile CPU designs, and find that existing mobile processors are ill-suited for mobile Web applications. The major sources of inefficiencies are the instruction delivery and data feeding.
(2) We customize microarchitectural parameters of existing designs, and propose two specialization techniques, SRU and Browser Engine Cache, that mitigate the instruction delivery and data feeding inefficiency of existing general-purpose designs.
(3) Combining customization and specialization techniques, we present WebCore, a general-purpose mobile CPU architecture substrate that is tailored for mobile Web applications, and achieves significant performance and energy-efficiency improvement over current mobile CPU designs.

The rest of the paper is organized as follows. We first provide a brief background of Web applications and the Web browser engine in Sec. 2. We describe our experimental setup including software/hardware infrastructure and application selection in Sec. 3. We then describe the design-space explorations that allow us to identify sources of inefficiency in existing general-purpose processors and customize them for mobile Web applications in Sec. 4. Building on top of the customized general-purpose designs, we further propose the two new specialization techniques in Sec. 5 and Sec. 6. We show that our proposed WebCore achieves significant performance and

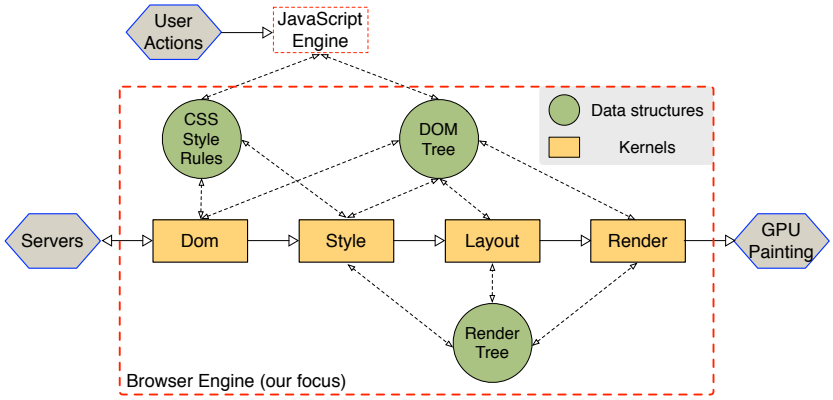


Fig. 1. Web browser overview.

energy-efficiency improvement over existing designs in Sec. 7. We review related work in Sec. 8, and conclude with future work in Sec. 9.

2 BACKGROUND ON WEB APPLICATIONS AND WEB BROWSER

In this section, we first provide a background of Web applications and the underlying Web browser architecture. We then focus on the computation kernels and their communication schemes in a Web browser. Such understanding helps us design effective customizations and specializations, such as those described in the later sections. Almost all modern Web browser engines fit into our description. Furthermore, we show each computation kernel's performance and energy breakdown to demonstrate the coverage of our study.

Web Applications Web applications are applications developed using Web languages including HTML, CSS, and JavaScript. To enable the "write-once, run-everywhere" feature of Web applications, the Web browser acts as a "virtual machine" that dynamically translates Web applications to different target platforms. Fig. 1 shows the overall flow of execution within any typical Web browser. In general, a Web browser has two core components: the rendering engine (e.g., Blink for Google Chrome and Gecko for Mozilla Firefox) and the JavaScript engine (e.g., V8 for Google Chrome and SpiderMonkey for Mozilla Firefox). The rendering engine translates HTML and CSS, and the JavaScript engine executes JavaScipt code.

It is important to emphasize that the rendering engine and the JavaScript engine are not completely isolated. Instead, JavaScript code may trigger rendering engine computations by modifying rendering engine data structures as we will discuss later. In this work, we focus on the rendering engine, including those executions that are triggered by JavaScript code. The JavaScript engine that involves the compiler, garbage collector, etc., is a separate issue beyond the scope of this work. Please refer to the Related Work section (Sec. 8) for a more elaborate discussion about JavaScript.

Computation Kernels The rendering engine mainly consists of four kernels: *Dom*, *Style*, *Layout*, and *Render*. The kernels, shown in boxes in Fig. 1, process Web application content and prepare pixels for a GPU to paint. Fig. 1 also shows the important data structures that the kernels consume. The DOM tree, CSS style rules, and Render tree are those important data structures, and they are heavily shared across the kernels. The data structures are shown in circles with arrows indicating information flow between the kernels.

The *Dom* kernel is in charge of parsing the webpage contents. Specifically, it constructs the DOM tree from the HTML files, and extracts the CSS style rules from the CSS files. Given the DOM tree and CSS style rules, the *Style* kernel computes the webpage's style information and stores the results in the Render tree. Each Render

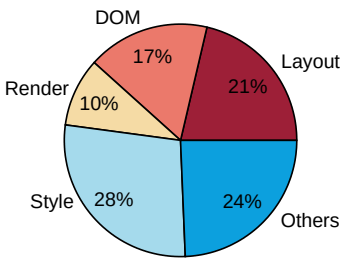


Fig. 2. Execution time breakdown of the browser's kernels.

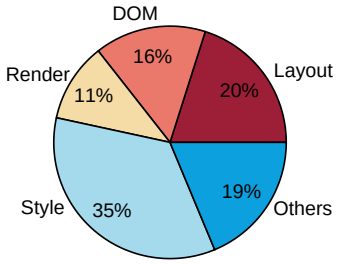


Fig. 3. Energy consumption breakdown of the kernels.

tree node corresponds to a visible element in the webpage. Once the style information of each webpage element is calculated, the *Layout* kernel recursively traverses the Render tree to decide each visible element's position information based on each element's size and relative positioning. The final $\langle x, y \rangle$ coordinates are stored back into the Render tree. Eventually, the *Render* kernel examines the Render tree to decide the z-ordering of each visible element so that they can be displayed in the correct overlapping order.

JavaScript code can trigger rendering engine kernels. For instance, a JavaScript function that implements an animated slideshow may change the source attribute of an HTML < img > element in order to display different images. Changing the source attribute modifies the DOM tree, and therefore triggers the *Layout* and *Render* kernels to display a new image. Different JavaScript codes can modify different rendering engine data structures, and therefore trigger different computation kernels. We refer interested readers to more detailed discussions of the rendering engine pipeline [48] for a better understanding.

Kernel Performance Fig. 2 shows the average execution time breakdown of the browser engine kernels. The measured data was gathered on a single-core Cortex-A15 processor in the Exynos 5410 SoC [70] while navigating the benchmarked Web applications using Google's open source Chromium browser [26]. The Exynos 5410 SoC is used in Samsung's Galaxy S4 smartphone, and is a representative modern mobile SoC. We will describe the benchmarked Web applications in Sec. 3. On average, the four kernels contribute to 76% of the total execution time. The *Style* kernel is the most time-consuming task. Please refer to Sec. 8 for discussion on multicores for Web browsing.

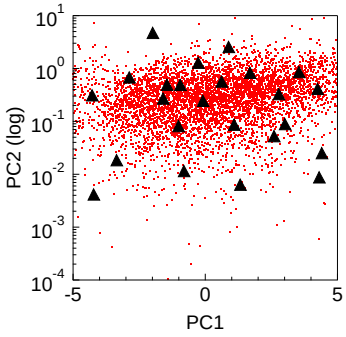
Kernel Energy Fig. 3 shows the average CPU energy consumption breakdown of the different Web browser engine kernels using the same experimental setup as above. We measure CPU power using National Instruments' X-series 6366 DAQ at 1,000 samples per second. Overall, the four kernels that we consider in this paper consume 81% of the total energy. In particular, The *Style* resolution kernel consistently consumes the most energy, typically around 35%.

3 EXPERIMENTAL SETUP AND VALIDATION

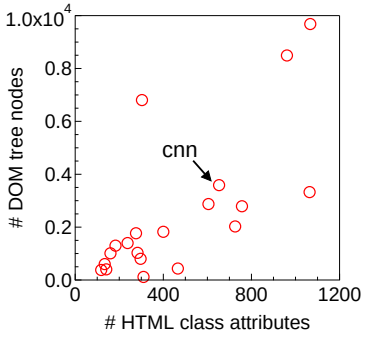
Before we begin our investigation, we describe our software infrastructure, specifically outlining our careful selection of representative webpages to study, and the processor simulator.

Web Browser We focus on the popular WebKit [81] rendering engine used in Google Chromium (Version 30.0) for our studies. WebKit is also widely used by other popular mobile browsers, such as Apple's Safari and Opera.

Benchmarked Web Applications We pay close attention to the choice of webpages to ensure that the WebCore design is not misled. We mine through the top 10,000 websites as ranked by Alexa [2] and pick the 12



(a) We pick 24 representative webpages from 10,000 of the hottest webpages as per `www.alexa.com`.



(b) `www.cnn.com` is a representative webpage from our benchmark suite because it is almost the centroid.

Fig. 4. Benchmark representativeness analysis.

most representative websites. All except one happen to rank among Alexa's top 25 websites. The 12 benchmarked websites also cover 10 of BBench's 11 webpages [29]. Please refer to Sec. 8 for a discussion of BBench. Sec. 7 lists the website names.

We consider not only the mobile version of the 12 websites, but also their desktop counterparts. Many mobile users still prefer desktop-version websites for their richer content and experience [13, 74]. Moreover, many mobile devices, especially tablets, typically load the desktop version of webpages by default. As webpage sizes exceed 1 MB [12], we must study mobile processor architectures that can process more complex content and not just simple mobile webpages.

We study 24 distinct webpages in total. The 24 benchmarked webpages are representative because they capture the webpage variations in both webpage-inherent and microarchitecture-dependent features. To prove this, we performed principal component analysis (PCA), which is a statistical method that reduces the number of inputs without losing generality [20]. PCA transforms the original inputs into a set of principal components (PC) that are linear combinations of the inputs. In our study, PCA calculates four PCs from about 400 distinct features. These four PCs account for 70% of the variance across all of the original 10,000 webpages. Fig. 4a shows the results for two major components, PC1 and PC2. IPC (microarchitecture-dependent feature) is the single most significant metric in PC1, and the number of DOM tree nodes (webpage-inherent feature) is the most significant metric in PC2. The triangular dots represent our webpages. They cover a very large spread of the top 10,000 webpages in the Internet.

Performance Metric We focus on the initial loading of Web applications. This is because user QoS experience is strongly tied to the initial load time in Web applications. For instance, it is estimated that 79% of online shoppers will not return to the website with slow load time [45].

Unless stated otherwise, we define Web application load time as the execution time that elapses until the `onload` event is triggered by the Web browser. It is worth noting that during the loading phase (i.e., before the `onLoad` event is triggered), many Web applications execute JavaScript code such as Ads and analytics. Therefore, our study not only takes into account the initial loading of the webpage, but also includes JavaScript activity that is triggered automatically by Web applications.

Simulators We assume the x86 instruction set architecture (ISA) for our study. Prior work shows that the ISA does not significantly impact energy efficiency for mobile workloads [14]. Therefore, we believe that our

microarchitecture explorations are generally valid across ISAs. We use Marss86 [64], a cycle-accurate simulator, in full-system mode to faithfully model all the network and OS activity. Performance counters from Marss86 are fed into McPAT [49] for power estimation.

We do our best-effort validation of the simulator by comparing it with an ARM platform because we do not have access to a measurable x86 mobile platform. We use the ODroid XU+E development board [33] that hosts the Exynos 5410 SoC as the hardware platform. The Exynos 5410 SoC is known for powering the Samsung's Galaxy S4 smartphone. The Exynos 5410 SoC contains an ARM Cortex-A15 processor. In our measurements, single core Cortex-A15 consumes 1.7 J energy and 2 seconds to load `www.cnn.com`. For comparison, we tune our simulator configurations to best match the microarchitecture parameters of Cortex-A15. The simulation results report 1.2 J and 2.2 seconds for energy consumption and loading time, respective.

4 CUSTOMIZING THE GENERAL-PURPOSE CORES

The industry has built both in-order (such as ARM Cortex A7 [3] and Intel Saltwell [38]) and out-of-order (such as ARM Cortex A15 [5] and Intel Silvermont [39]) cores for mobile processors. By exploring the vast design space by varying design parameters (Sec. 4.1), we find that the out-of-order designs provide more flexibility for energy versus performance trade-offs than in-order designs (Sec. 4.2). Within the out-of-order design space, we further observe that existing mobile processor configurations contain inherent sources of energy inefficiency in instruction delivery and data feeding. We customize the general-purpose cores by tuning corresponding design parameters to mitigate these inefficiencies (Sec. 4.3). The customization step also allows us to derive a better general-purpose baseline for further specialization.

4.1 Design Space Exploration

Design Space Specification We define the set of tunable microarchitectural parameters in Table 1. We restrict each parameter's range to limit the total exploration space. For example, we restrict the values of functionally related parameters from reaching a completely unbalanced design [15]. For example, the number of functional units increases with the issue width so that the execution engine does not become the processor's bottleneck. In addition, we do not consider single-issue out-of-order processors, which are known to be energy inefficient. Overall, we consider over 3 billion designs.

We intentionally relax the design parameters beyond the current mobile systems in order to allow an exhaustive design space exploration. For example, we consider up to 128 KB L1 cache design whereas most L1 caches in existing mobile processors are 32 KB in size. Also, We eliminate overly aggressive designs with more than 2 W TDP, which is a conservative estimation of mobile CPU TDP [31]. We assume a fixed core frequency in our design-space exploration. We use 1.6 GHz, a common value in mobile processors, to further prune the exploration space. It is worth noting that the latency of both the L1 and L2 caches can still vary, and therefore we include different cache designs in the exploration space.

We use a constant memory latency to model the memory subsystem because we do not observe significant impact of the memory system on the mobile Web browsing workload. According to hardware measurements on the Cortex-A15 processor using ARM's performance monitoring tool Streamline [6], the MPKI for the L2 cache across all the webpages is below 5. We observe similar low L2 MPKI, i.e. low main memory pressure, in our simulations. Therefore, we use a simpler memory system to further trim the search space. However, it is also important to note that our processor model can be readily integrated with the various analytical memory models [59] or simulators [78] if a more detailed memory system is required for analysis.

Statistical Inference Method Because we consider billions of design points, it is not feasible to simulate all of them owing to time constraints. Therefore, we leverage the statistical inference technique that trains predictive models using a small number of samples. Such models reflect how different microarchitecture parameters, both

Table 1. Microarchitecture design-space parameters (i::j::k denotes values ranging from i to k at steps of j)

Parameters	Measure	Range
Issue width	count	1::1::4
# Functional units	count	1::1::4
Load queue size	# entries	4::4::16
Store queue size	# entries	4::4::16
Branch prediction size	log_2(#entries)	1::1::10
ROB size	# entries	8::8::128
# Physical registers	# entries	5::5::140
L1 I-cache size	log_2(KB)	3::1::7
L1 I-cache delay	cycles	1::1::3
L1 D-cache size	log_2(KB)	3::1::7
L1 D-cache delay	cycles	1::1::3
L2 cache size	log_2(KB)	7::1::10
L2 cache delay	cycles	16,32,64

Table 2. Performance (IPC) and power prediction errors of our microarchitecture design-space exploration using statistical inferential models

Kernel	IPC Error (%)	Power Error (%)
Dom	7.98	5.64
Style	6.04	4.52
Layout	5.47	4.94
Render	5.68	4.37
Mix	5.47	4.76

individually and collectively, influence performance and power consumption. Statistical inference methods have been used successfully in the past for architecture design-space exploration [28, 47].

In particular, we use linear regression modeling [34] to construct our predictive models. A linear regression model can be formulated as in Equ. 1, where y denotes the response, $x = x_1, ..., x_p$ denote p predictors, and $\beta = \beta_0, ..., \beta_p$ denote corresponding coefficients of each predictor. The *least squares method* is used to solve the regression model by identifying the best-fitting β that minimizes the residual sum of squares (RSS) [35]. In our case, the response y is either performance (measured in terms of instruction per cycle, IPC) or power, and the predictors x_i are microarchitecture structures listed in Table 1.

$$y = \beta_0 + \sum_{i=1}^{p} x_i \beta_i \tag{1}$$

We find that 2,000 *uniformly at random* (UAR) samples of microarchitecture configurations from the design space are sufficient in our case to construct robust models. We also obtain 500 additional UAR samples from the cache design space (both L1 and L2) to reinforce the credibility of instruction and data cache design predictions.

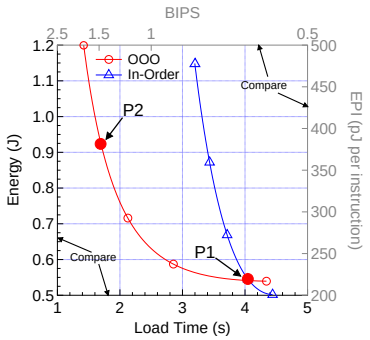


Fig. 5. In-order versus out-of-order Pareto optimal frontiers.

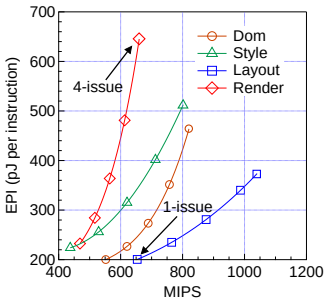


Fig. 6. In-order Pareto optimal frontier for each kernel.

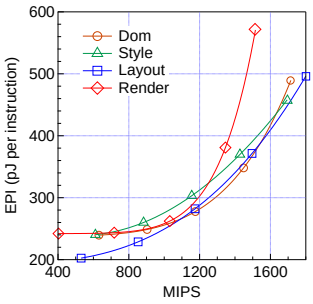


Fig. 7. Out-of-order Pareto optimal frontier for each kernel.

We perform cross-validation of the model (i.e., we partition a sample dataset into complementary subsets, and perform analysis on one subset and validate the analysis on the other subset), and then obtain additional samples from the design space for full evaluation.

In order to derive general conclusions about the design space and optimize for the common case, in this section we present only our in-depth analysis for the representative website `www.cnn.com`. Fig. 4b compares `www.cnn.com` with other webpages to demonstrate that it is indeed representative of the other benchmarked webpages. The x-axis and y-axis represent the number of DOM tree nodes and the number of class attributes in HTML. These are the two webpage characteristics that are most correlated with a webpage's load time and energy consumption [88]. As the figure shows, `www.cnn.com` is roughly the centroid of the benchmarked webpages, and thus we use it as a representative webpage for the common case.

We construct the performance and power models for the four kernels described in Sec. 2, as well as the entire Web browser rendering engine. We construct predictive models for out-of-order and in-order design space separately because microarchitecture structures have different impact on performance and power in in-order and out-of-order pipelines. Table 2 shows the out-of-order model error rate for predicting both performance and power consumption. In general, the out-of-order models' error rates are below 6.0%. The in-order models (not shown) are more accurate because of their simpler design. On average, the in-order performance and power models' errors are within 5% and 2%, respectively.

4.2 In-order vs. Out-of-order Design Space Exploration

In this section, we explore both the in-order and out-of-order space to identify the optimal general-purpose design for the entire browser engine. We find that out-of-order cores can better balance performance with energy, and are therefore better designs for mobile Web browsing. In order to understand the fundamental reasons, we study the individual Web browser engine kernels and demonstrate that the out-of-order logic can cover the variances across the different kernels through its complex execution logic. In contrast, in-order designs either overestimate or underestimate the hardware requirements.

Entire Rendering Engine Design space exploration helps customization at the "macro-architecture" level, i.e., determining between in-order and out-of-order designs. We understand the difference between in-order and out-of-order design space by examining their Pareto optimal frontiers. Design points on a Pareto optimal frontier reflect different optimal design decisions given specific performance/energy targets. The Pareto-optimal is more general than the (sometimes overly specific) EDP, ED^2P metrics, etc. Design configurations optimized for such metrics have been known to correspond to different points on the Pareto-optimal frontier [7].

Fig. 5 shows the Pareto-optimal frontiers of both in-order and out-of-order designs between energy and performance. We use energy per instruction (EPI) for the energy metric, and million instructions per second (MIPS) as the performance metric.

We make two important observations from Fig. 5. First, the out-of-order design space offers a much larger performance range (∼1 BIPS between markers P1 and P2, see top x-axis) than the in-order design space (< 0.5 BIPS), which reflects the out-of-order's flexibility in design decisions. Second, the out-of-order design frontier is flatter around the 4-second webpage load time range (see marker P1) than in the in-order design, which indicates that the out-of-order design has a much lower marginal energy cost. The observation indicates that processor architects can make design decisions based on the different performance goals without too much concern about the energy budget. In contrast, the in-order design space quickly enters the region of diminishing returns (i.e., sharp increase in energy consumption) as we push toward webpage load times that are less than 4 seconds. In other words, the in-order design has a low marginal performance value (or equivalently high marginal cost of energy).

To understand the major limitation of the in-order design, we compare the microarchitecture configurations of the in-order and out-of-order designs at the crossover point P1 in Fig. 5.The "P1:OoO" and "P1:InO" columns in Table 3 list the out-of-order and in-order configurations, respectively. Even though both designs achieve the same performance, the in-order design has much larger L1 and L2 cache sizes. Therefore, the in-order design at P1 provides better instruction delivery and data feeding than the out-of-order design. Thus, we conclude that it is the inability of the *execution logic* of in-order designs that inhibits better performance than its out-of-order counterpart.

Examining the detailed microarchitecture configurations of design points beyond the P1 crossover point (i.e., < 4 seconds), we further find that the in-order design quickly shifts toward a 4-wide issue with a much larger L2 cache. However, such designs have a very high marginal energy cost, which can lead to energy-inefficient designs as compared to their corresponding out-of-order counterparts.

Individual Kernels To further understand why the in-order design is unsuitable for Web browser workloads, we study the individual kernels' behavior. Fig. 6 and Fig. 7 show the Pareto-optimal frontiers of the in-order and out-of-order design space for each Web browser engine kernel. The kernel behavior is remarkably different across the two design spaces. In the in-order design, the kernel trade-offs are sharper, and more distinct from one another. For example, to achieve the same performance level at 800 MIPS, the EPI difference between the *Style* and *Layout* kernels is ∼300 pJ. In contrast, the difference is minimal (< 50 pJ) in the out-of-order design space.

Because the kernel difference in the in-order designs is more pronounced than in the out-of-order designs, we conclude that the different kernels require different in-order designs for a given fixed-performance goal. As we push toward more performance in the in-order design space, some kernels stop scaling gracefully on the energy versus delay curve. For example, among the four kernels, only the *Layout* kernel scales well beyond 850 MIPS. In contrast, the *Render* kernel's MIPS range is severely limited between 460 MIPS and 650 MIPS. Since all kernels are on the critical path of webpage load performance, the kernels that do not scale gracefully quickly become critical performance bottlenecks, which results in the low marginal performance improvement for the entire Web browser engine.

As an example for the kernel variance in the in-order designs, our analysis shows that to achieve 650 MIPS, the *Render* kernel requires 4-wide issue, whereas the *Layout* kernel only requires single-issue. The reason for the large issue width difference is that the *Layout* kernel mainly performs floating-point (FP) operations in order to calculate the actual (x, y) coordinates of webpage elements. In contrast, the *Render* kernel mostly performs control-flow instructions traversing trees without too much computation. The high FP intensity in the *Layout* kernel results in better instruction-level parallelism (ILP), which is easily exploited by a single-issue machine. Prior work has shown that FP programs typically have higher instruction retirement rate because they have larger dependency distances between instructions [66, 83].

Table 3. Microarchitecture configurations for the selected design points in Fig. 5. They represent different energy-delay trade-offs. For comparison purpose, we also show the corresponding microarchitecture parameters for ARM Cortex-A15, whose information is gathered from measurements using the 7-Zip LZMA Benchmark [1] and ARM's public presentation [4]. Question marks indicate Cortex-A15 information that is not publically available

	P1:OoO	**P1:InO**	**P2**	**Cortex-A15**
Issue width	1	2	3	3
# Functional units	2	2	3	8
Load queue size (# entries)	4	N/A	16	16
Store queue size (# entries)	4	N/A	16	16
BTB size (# entries)	1024	1024	128	64
ROB size (# entries)	128	N/A	128	40+
# Physical registers	128	N/A	140	N/A
L1 I-cache size (KB)	64	128	128	32
L1 I-cache delay (cycles)	1	2	2	N/A
L1 D-cache size (KB)	8	64	64	32
L1 D-cache delay (cycles)	1	1	1	4
L2 cache size (KB)	256	1024	1024	512~4096
L2 cache delay (cycles)	16	16	16	21

In contrast, such pronounced inter-kernel variance is not present in out-of-order designs. For example, at 1,200 MIPS, which is the "*knee of the curve(s)*," all the kernels have similar EPIs. This is because at 1,200 MIPS, all kernels have similar microarchitecture structure configurations. Table 4 shows the optimal microarchitecture configurations of the four kernels at 1,200 MIPS. We observe that the microarchitecture configurations of different kernels are similar. For example, all the kernels require a large number of physical registers to resolve dependencies, and none of the kernels need the widest issue width (i.e., 4-wide). The similar microarchitecture configurations of different kernels in the out-of-order design space indicates that out-of-order cores can explore the ILP for each kernel without bias, and therefore enable wider energy-performance trade-off space.

4.3 Energy Inefficiency in the Customized Core Designs

Although the design space exploration of general purpose cores allows us to flexibly trade performance and energy with each other for the mobile web browsing workload, we find that they still contain inherent sources of energy-inefficiencies. In this section, we show that instruction delivery and data feeding are the most sensitive components to energy efficiency in the out-of-order design.

We broke down the processor microarchitecture investigation into its three high-level components: instruction delivery, data feeding, and execution engine [65]. All of these components must be optimized in order to achieve a balanced design that does not suffer from performance and energy bottlenecks.

We focus on out-of-order designs because in-order designs are merely more effective in extremely low performance region. In particular, we examine two specific optimization points (i.e., P1 and P2) in Fig. 5 that represent optimized designs for different performance and power goals. P1 is an out-of-order design optimized for minimal energy consumption. P2 focuses on minimal energy consumption at 1500 MIPS. 1500 MIPS corresponds to an approximate webpage load time of 1.5 seconds.

Table 4. Pareto-optimal microarchitecture configuration of out-of-order designs for each rendering engine kernel at 1,200 MIPS. We find that the optimal designs for different kernels have similar microarchitecture configurations, leading to similar energy-performance trade-off as shown in Fig. 7.

	Dom	**Style**	**Layout**	**Render**
Issue width	2	3	2	2
# Functional units	2	3	2	2
Load queue size (# entries)	16	12	16	12
Store queue size (# entries)	16	12	16	12
Branch prediction size (# entries)	512	2	1024	2
ROB size (# entries)	128	128	128	128
# Physical registers	128	130	140	140
L1 I-cache size (KB)	128	128	64	64
L1 I-cache delay (cycles)	2	2	1	1
L1 D-cache size (KB)	32	128	64	8
L1 D-cache delay (cycles)	1	2	1	1
L2 cache size (KB)	128	1024	256	128
L2 cache delay (cycles)	16	16	16	16

The "P1:OoO" and "P2" columns in Table 3 summarize the microarchitecture parameters. For comparison purpose, we also show the corresponding microarchitecture parameters for ARM Cortex-A15, whose information is gathered from measurements using the 7-Zip LZMA Benchmark [1] and ARM's public presentation [4]. Overall, as the design objective biases more toward performance (shifting from P1 to P2), the microarchitecture configuration tends to be more complex and aggressive. More importantly, we find that the P1 and P2 configurations are different from current mobile processor designs in instruction delivery and data feeding aspects. Let us explain our findings.

Instruction Delivery Delivering instructions for execution is a major issue in both P1 and P2 designs. For example, current mobile processors have a small L1 instruction/data cache that is typically 32 KB in size. However, both P1 and P2 require a much larger L1 instruction cache of at least 64 KB. In addition to large instruction cache size, both P1 and P2 designs also require a large ROB as compared to what is typically found in existing mobile CPU (128 versus 40) to provide a large instruction window. Although a 64 KB L1 instruction cache and a 128 entry ROB are already excessively large for mobile CPU designs, the effective microarchitecture structure in real system would be even smaller due to resource contention from other concurrent applications.

Data Feeding Delivering data for computation is not a bottleneck when we optimize for energy (i.e., P1). However, they become critical as we shift the design goal toward performance (i.e., P2). Optimizing for P2 in Fig. 5 necessitates a 64 KB data cache. It achieves a low miss ratio of only 7.7%. Similar to instruction delivery, a large data cache is also more favorable to energy efficiency than having than a large L2 cache. The reasons for a large data cache are twofold. First, processing webpages typically involves a large footprint on the principal data structures (Sec. 2). For example, profiling results show that the average data reuse distance for DOM tree accesses (excluding other memory operations interleaved with DOM accesses) is about 4 KB. Second, different kernels are interleaved with each other during execution, which increases the effective data reuse distances of the important data structures.

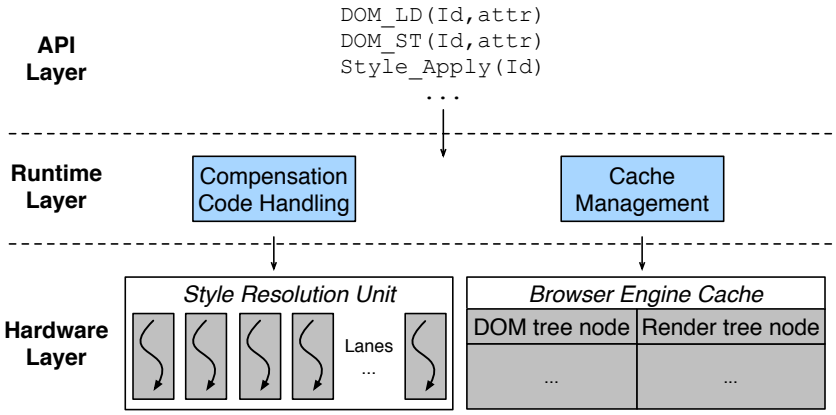


Fig. 8. Hardware-software co-design framework.

In summary, unusual design parameters in a processor core tuned for the mobile Web applications indicate that both instruction delivery and data feeding are critical to guarantee high performance while still being energy efficient. Our study motivates specialization techniques to offset instruction and data supply overhead.

5 STYLE RESOLUTION UNIT

We propose specialized hardware mechanisms to mitigate the instruction delivery and data feeding inefficiencies in the customized out-of-order core designs. In particular, we introduce two new hardware structures: a Style Resolution Unit (SRU) and a Browser Engine Cache (BEC). This section focuses on the SRU and the next section focuses on the BEC.

The SRU is an accelerator for the critical *Style* kernel within the Web browser rendering engine. The SRU design is based on the observation that the *Style* kernel has abundant fine-grained parallelism that is hidden in a software implementation but can be captured by a dedicated hardware structure (Sec. 5.1). To exploit the inherent fine-grained parallelism, the SRU employs a multi-lane parallel architecture, which greatly reduces the instruction delivery overhead. To reduce the data feeding pressure, the SRU is tightly coupled with a small scratchpad memory that brings operands closer to the SRU (Sec. 5.2).

To maintain general-purpose programmability, these new hardware structures are accessed via a set of high-level language APIs. The APIs are implemented through a runtime library with only slight modification to the current browser implementation (Sec. 5.3). Fig. 8 shows an overview of our proposed optimization framework. Overall, the hardware supports fast and energy-efficient execution and data communication, and the library hides the hardware complexity, manages the hardware executions, and eases the software development effort.

5.1 Motivation

Optimizing the *Style* kernel would improve the overall energy efficiency the most for the following reasons. The *Style* kernel is the most time-consuming task in the rendering engine. As shown in Fig. 2, it consumes 35% of the total rendering engine execution time. It also dominates the energy consumption by consuming 40% of the total energy as shown in Fig. 3.

In order to mitigate the instruction delivery and data communication overhead of the *Style* kernel, we propose a special functional unit called the *Style Resolution Unit* (SRU) that is tightly coupled with a small scratchpad

```
1   // matching phase
2   matchedRules = matching(DOMTree, DOMNodeId, CSSRules);
3
4   // applying phase
5   foreach (rule in matchedRules) {
6     foreach (property in rule) {
7       switch (property.ID) {
8         case Font:
9           RenderStyle[Font] = FontHandler(property, DOMNodeId);
10          break;
11        case Color:
12          RenderStyle[Color] = ColorHandler(property, DOMNodeId);
13          break;
14        …
15        case N: ...
16      }
17    }
18  }
```

Fig. 9. Pseudo-code of the *Style* kernel. It consists of a matching phase and an applying phase. SRU accelerates the applying phase, which takes about two-thirds of the *Style* kernel execution time.

memory. The SRU exploits fine-grained parallelism to reduce the amount of instructions and potential divergences. The scratchpad memory reduces data communication pressure by bringing operands closer to the SRU.

The *Style* kernel consists of two phases: a matching phase and an applying phase. Fig. 9 shows the pseudo-code of the two phases. Previous work [16, 56] focuses on parallelizing the matching phase. However, in our profiling, we find that the applying phase takes nearly twice as long to execute as the matching phase. Therefore, we focus on the applying phase. The applying phase takes in a set of CSS rules (`matchedRules`) as input, iterates over each rule in the correct cascading order [77] to calculate each style property's final value (e.g., the exact-color RGB values, font width pixels). The final values are stored back to the Render tree (the `RenderStyle` array).

The key observation we make in the applying phase is that there are two types of inherent parallelism: "rule-level parallelism" (RLP) and "property-level parallelism" (PLP). Improving the energy efficiency of the *Style* kernel requires us to exploit both forms of parallelism in order to reduce the control-flow divergence and data communication overheads. Our profiling results indicate that both control flow and memory instructions put together constitute 80% of the total instructions that are executed within the *Style* kernel.

RLP comes from the following. In order to maintain the correct cascading order, each rule contained in the input data structure must be sequentially iterated from the lowest priority to the highest, so that the higher-priority rules can override the lower-priority rules. However, in reality, we could speculatively apply the rules with different priorities in parallel, and select the one with the highest priority. PLP follows RLP. Each rule has multiple properties, and each property is examined by the engine to set the corresponding data field in the Render tree according to its property ID. Because properties are independent of one another, handling of their processing routines can be dealt with in parallel.

It is worth noting that the RLP and PLP are not easily captured by SIMD hardware in existing processors. This is because the computations associated with different properties (e.g., `FontHandler` and `ColorHandler` in Fig. 9) are different, and therefore they do not conform with the "single instruction, multiple data" execution pattern.

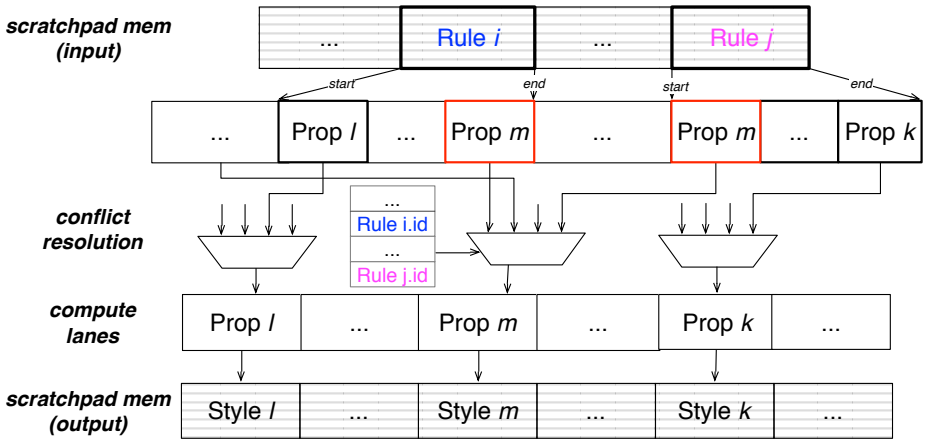


Fig. 10. SRU coupled with scratchpad memories.

5.2 Hardware Design

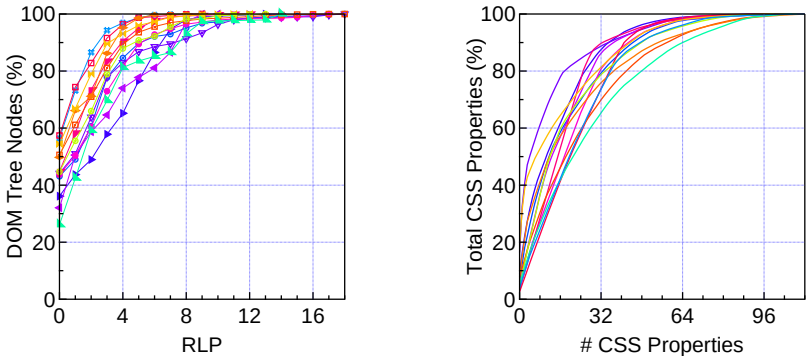
We propose a parallel hardware unit that exploits both RLP and PLP, called the Style Resolution Unit. The SRU aggregates enough computations to reduce control-flow divergences and increase arithmetic intensity. It is accompanied by data storage units for both input and output. Note that it is not easy to exploit software-level parallelism for PLP and RLP because of the complex control flow, memory aliasing, and severe loop-carried dependencies.

In addition, we noticed that the input to the applying phase, `matchedRules`, is an intra-kernel shared data structure between the matching and applying phases. Storing such short-lived data into the memory hierarchy, and accessing it through traditional load and store instructions, results in slow computation. It also wastes energy. Therefore, we provide a scratchpad memory for the input. Similarly, we store the output structure (i.e., `RenderStyle`) in a separate scratchpad memory.

Fig. 10 shows the structure of the SRU with scratchpad memory for input and output data. SRU has multiple lanes, with each lane dealing with one CSS property. Assume Rule i and Rule j are two rules from the input that are residing in the scratchpad memory. Rule i has higher priority than Rule j. Prop l and Prop m are two properties in Rule i. Similarly, Rule j has properties Prop k and Prop m. Prop l and Prop k can be executed in parallel using different SRU lanes because they do not conflict with each other. However, Prop m is present in both rules, and as such it causes an SRU lane conflict, in which case the MUX selects the property from the rule with the highest priority, which in our example is Rule i.

Design Considerations A hardware implementation can have only a fixed amount of resources. Therefore, the number of SRU lanes and the size of the scratchpad memory is limited. Prior work [88] shows that the number of matched CSS rules and the number of properties in a rule can vary from one webpage to another. As such, a fixed design may overfeed or underfeed the SRU if the resources are not allocated properly.

We profile the webpages to determine the appropriate amount of resource allocation required for the SRU. Profiling indicates that 90% of the time, the RLP is below or equal to 4 (Fig. 11a). Therefore, our design's scratchpad memory only stores up to four styles. Similarly, 32 hot CSS properties cover about 70% of the commonly used properties (Fig. 11b). Thus, we implement a 32-wide SRU where each lane handles one hot CSS property. Due to these considerations, the input and output scratchpad memories are each 1 KB in size.



(a) RLP analysis. (b) CSS property analysis.

Fig. 11. Analysis of RLP and CSS properties across webpages.

Furthermore, not all of the properties are delegated to the SRU. For example, some style properties require information on the parent and sibling nodes. To avoid complex hardware design for recursions and loops with unknown iterations, we do not implement them in our SRU prototype. The runtime library performs these checks, which we discuss later in Sec. 5.3. Despite the trade-offs we make, about 72.4% of the style rules across all the benchmarked webpages can utilize the SRU.

5.3 Software Support and Programmability

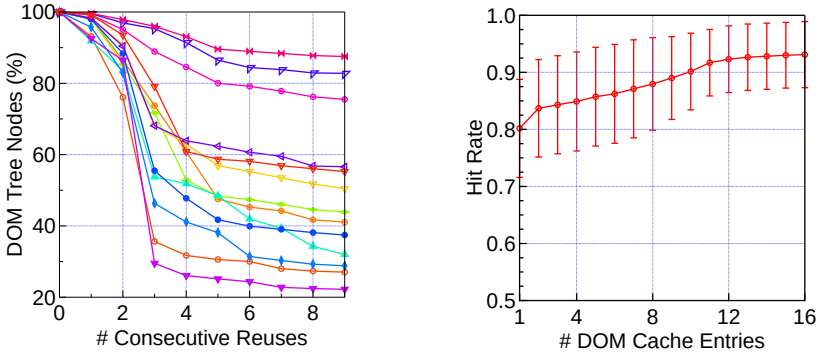
The SRU can be accessed via a small set of instruction extensions to the general-purpose ISA. In order to abstract the low-level details away from application developers, we provide a set of library APIs in high-level languages. Application developers use the APIs without knowing the existence of the specialized hardware. It is important to notice that these software APIs are used by Web browser rendering engine developers rather than high-level Web application developers. WebCore does not affect the programming interface of Web application developers, and therefore has no impact on the Web application development productivity.

```
1  // matching phase
2  matchedRules = matching(DOMTree, DOMNodeId, CSSRules);
3
4  // applying phase
5  Style_Apply(DOMNodeId, matchedRules);
```

Fig. 12. Pseudo-code of the *Style* kernel with the new API.

Programmers trigger the style resolution task by issuing a `Style_Apply(Id, Rules)` API, in which `Id` represents a DOM tree node ID and `Rules` represents matched CSS rules produced by the matching phase. Fig. 12 illustrates the pseudo-code of the *Style* kernel using the provided API. Comparing against the original code in Fig. 9, we notice that the matching phase is not changed while the applying phase is greatly simplified with the `Style_Apply` API.

One key task of this API implementation is to examine all the CSS properties of a particular DOM node because not all the CSS properties are implemented in the SRU (as discussed in Sec. 5.2). For properties that can be



(a) DOM node reuse behavior.

(b) DOM node access hit rate.

Fig. 13. DOM tree access behavior across webpages.

offloaded to the SRU, the API implementation loads related data into the SRU's scratchpad memory. For those "unaccelerated" properties, the runtime creates the necessary compensation code. Specifically, we propose relying on the existing software implementation as a fail-safe fallback mechanism. Once the style resolution results are generated, the results can be copied out to the output scratchpad memory.

6 BROWSER ENGINE CACHE

To further improve the energy-efficiency of date feeding, we propose the browser engine cache. It is based on the observation that Web applications' accesses to principal data structures, such as the DOM tree and the Render tree, exhibit heavy data reuse and predictable access pattern (Sec. 6.1). Based on such an observation, the browser engine cache uses a small hardware memory structure coupled with a lightweight software-based cache management layer to provide energy-efficient data access (Sec. 6.2). In addition, similar to SRU, we also provide a set of high-level language APIs that allow Web browser developers to easily access the browser engine cache (Sec. 6.3).

6.1 Motivation

The DOM tree and Render tree are the two most important data structures because they are shared across different kernels, as shown in Fig. 1. We propose the Browser Engine Cache to improve the energy-efficiency of accessing them. Specifically, the browser engine cache consists of a DOM cache and a Render cache for the DOM tree and Render tree, respectively. We use the DOM to explain our locality observation. Similar analysis and design principles also apply to the render cache. Note that the browser engine cache focuses on improving the energy efficiency of data feeding. We will discuss techniques for improving the performance aspect of data accesses in Sec. 8.

The energy inefficiency of the traditional cache is best embodied in the performance-oriented design P2 in Table 3. P2 requires a larger data cache (64 KB) compared to a traditional mobile core. Although a large cache achieves a high hit rate of 93%, it leads to almost one-fourth of the total energy consumption. However, through careful characterizations, we find that accesses to the DOM/Render tree have strong locality and regular access pattern such that they can benefit from a small and energy-efficient cache memory, rather than the large power-hungry traditional caches. Let us explain our observations below.

First, we find that data accesses to the DOM tree have heavy reuses. Fig. 13a shows the cumulative distribution of DOM tree node reuse. Each (x, y) point corresponds to a portion of DOM tree nodes (y) that are consecutively reused at least a certain number of times (x). About 90% of the DOM tree nodes are consecutively reused at least two times (three times in total), which reflects strong data locality. This indicates that a very small cache can achieve the similar hit rate as a regular cache, but with much lower power.

Second, we find that the accesses to the DOM tree have regular stream-like patterns. To illustrate this, Fig. 14 shows two representative data access patterns to the DOM tree from `www.sina.com` and `www.slashdot.org`. Each (x, y) point is read as follows. The x-th access to the DOM tree operated on the y-th DOM node. We observe a common streaming pattern. Such a streaming pattern is due to the intensive DOM tree traversal that is required by many rendering engine kernels. For example, in order to match CSS rules with descendant selectors such as "div p," which selects any <p> element that is a descendant of in the DOM tree, the *Style* kernel must traverse the DOM tree, one node at a time, to identify the inheritance relation between two nodes. Similarly, the *Layout* kernel must traverse the Render tree (recursively) to determine the size of each webpage element, which in turn depends on the sizes of the elements contained within it.

In summary, the rendering engine typically operates on one DOM tree node heavily and traverses to the next one. After the rendering engine moves past a DOM node, it is rarely re-referenced soon. Such a unique access behavior motivates the browser engine cache design as we describe below.

6.2 Hardware Design

We propose the DOM cache to capture the DOM tree data locality. It sits between the processor and the L1 cache, effectively behaving as an L0 cache. Each cache line contains the entire data for one DOM tree node, which is 698 bytes in our design. Different from the data array in a regular cache, we implement each cache entry (both in the DOM cache and render cache) as a collection of registers instead of a wide cache line. Each register holds one attribute of the DOM (Render) tree node, and can be individually accessed through special memory instructions from the software.

The motivations to split each DOM cache line into individually addressable registers are as follows. First, not all the attributes of a node are accessed every time a node is referenced such that pre-loading all the node data from L1 cache to the browser engine cache lead to performance and energy penalty. For example, a Render tree node most often is of either `RenderBlock` or `RenderInline` type, each of which involves its own set of attributes. The browser can decide what attributes to load depending on what type a Render tree node is. Second, splitting the large memory array into small registers also allows fast and more energy-conserving accesses.

We choose to implement the DOM cache as a "software-managed" cache–i.e., the data is physically stored in hardware memory, and the software performs the actual cache management, such as insertion and replacement. Prior work has demonstrated effective software-managed cache implementations [30]. It is possible to implement the DOM cache entirely in hardware, similar to a normal data cache. Our motivation for a software-managed cache is to avoid the complexity of a hardware cache. Typically, the cache involves hardware circuitry whose overhead can be high, especially for extremely small cache sizes.

The software overhead for the software-managed browser cache is relatively insignificant for the following reasons. First, a simple replacement policy that always evicts the earliest inserted line is sufficient. Due to the streaming pattern shown in Fig. 14, DOM tree nodes are rarely re-referenced soon after the browser engine moves past them. Therefore, a simple FIFO design is almost as effective as the least recently used policy, but with much less management overhead.

Second, a very small number of DOM cache entries guarantee a high hit rate. Therefore, the cache-hit lookup overhead is minimal. Fig. 13b shows how the hit rate changes with the number of entries allocated for the DOM tree. The curve represents the average hit rate, and the error bars represent the standard deviations across different webpages. Across all the webpages, a 4-entry design can achieve about 85% hit rate, and so we use this

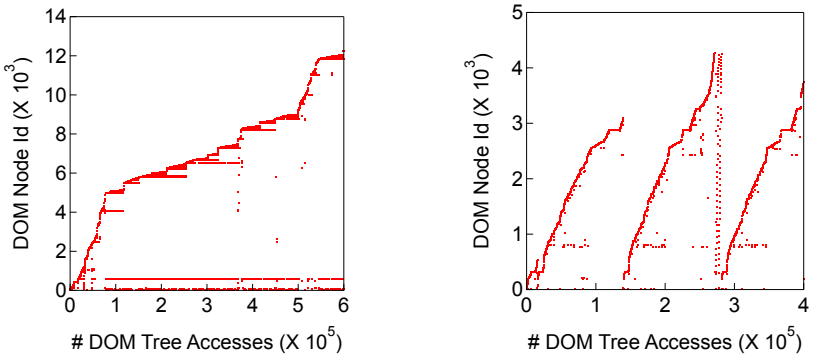


Fig. 14. Representative DOM tree access patterns.

configuration. In this sense, the DOM cache is effectively a single set, 4-way fully associative cache. Similarly, the render cache contains two entries (i.e., two cache lines). On average, it achieves over 90% hit rate.

6.3 Software Support and Programmability

To access a particular DOM tree node in the rendering engine, developers issue `DOMCache_LD(Id, attr)` and `DOMCache_ST(Id, attr, data)` for read and write operation, respectively. Similar APIs are also provided for the Render Cache. In the provided APIs, `Id` represents the DOM tree node ID (similar to the `Style_Apply()` API), `attr` represents a particular DOM node attribute, and `data` indicates the new data of the specified `attr`. Recall that our DOM cache design allows each attribute of a DOM node to be individually addressed (Sec. 6.2). The syntax of both APIs allow developers to fully utilize this feature.

```
1   Class Attribute {
2   public:
3     void setValue(const AttrVal &value) {
4       m_value = value;
5     }
6
7   private:
8     AttrName m_name;
9     AttrVal m_value;
10  }
```

Original Code

```
Class Attribute {
public:
  void setValue(const AttrVal &value) {
    DOMCache_ST(toNodeID(), m_name, value);
  }

private:
  AttrName m_name;
  AttrVal m_value;
}
```

New Code

Fig. 15. Using the `DOMCache_ST()` API in the rendering engine. The new DOM attribute store API replaces the original attribute value assignment, and performs cache manegement.

Fig. 15 shows how `DOMCache_ST()` API is used in the rendering engine. It is used to set value of any given attribute in the `setValue()` method of the `Attribute` class. Specifically, `DOMCache_ST()` replaces the original value assignment. The API implementation performs the actual hardware memory accesses as well as cache management, such as replacement and insertion. For example, the API needs to maintain an array, similar to the tag array in a regular cache, to keep track of which DOM nodes are in the cache and whether they are modified.

Effectively, the runtime library of DOM cache APIs implements a cache simulator. However, the runtime overhead is negligible due to the simple cache design as described in Sec. 6.2.

It is worth noting that using DOM cache APIs only affects the primitive classes of a rendering engine (such as the `Attribute` class in Fig. 15) while maintaining the interface between primitive classes and the rest of the rendering engine unchanged. For example, rendering engine developers can still use the same `setValue()` method to update an attribute's value. Therefore, we do not expect using the new APIs to affect the development productivity.

7 WEBCORE EVALUATION

In this section, we first present the power and timing overhead analysis of the proposed specialization techniques (Sec. 7.1). We then evaluate the energy-efficiency implications of the SRU and the browser engine cache individually (Sec. 7.2, Sec. 7.3). In the end, we show the energy-efficiency improvement combining both customization and specialization (Sec. 7.4). In particular, we show that our specializations can achieve significantly better energy efficiency than simply dedicating the same amount of area and power overhead to tune the conventional general-purpose cores.

We evaluate our optimizations against three designs, D1 through D3. D1 refers to the energy-conscious design (P1) that we explored in Fig. 5. Similarly, D2 refers to the performance-oriented design (P2) in Fig. 5. D3 mimics the common design configuration of current out-of-order mobile processors. We configure D3 as a three-issue out-of-order core with 32-entry load queue and store queue, 40 ROB entries, and 140 physical registers. It has a 32 KB, 1-cycle latency L1 data and instruction cache, and a 1 MB, 16-cycle latency L2 cache.

7.1 Overhead Analysis

We use CACTI v5.3 [76] to estimate the memory structures overhead. We implement the SRU in Verilog and synthesize our design in 28 nm technology using the Synposys toolchain.

Area The size of SRU's scratchpad memory is 1 KB. The DOM cache size is 2,792 bytes. The render cache size is 1,036 bytes. The hardware requirements for the SRU are mainly comparators and MUXes to deal with control flow, and simple adders with constants inputs to compute each CSS property's final value. In total, the area overhead of the memory structures and the SRU logic is about 0.59 mm^2, which is negligible compared to typical mobile SoC size (e.g., Samsung's Exynos 5410 SoC has a total die area size of 122 mm^2 [84]).

Power The synthesis reports that the SRU logic introduces 70 mW total power under typical stimuli. The browser engine cache and the SRU scratchpad memory add 7.2 mW and 2.4 mW to the dynamic power, respectively. They are insignificant compared to power consumption for Web browsing (in our measurements, a single core Cortex-A15 consumes about 1 W for webpage loading). Clocking gating can reduce the power consumption further [50]. But we are conservative in our analysis and do not assume such optimistic benefits.

Timing Both the browser engine cache and SRU scratchpad memory can be accessed in one cycle, which is the same as the fastest L1 cache latency in our design space. The synthesis tool reports that the SRU logic latency is about 16 cycles under 1.6 GHz. Later in our performance evaluation, we conservatively assume the SRU logic is not pipelined.

Software The software overhead mainly includes cache management and SRU compensation code creation. The overhead varies depending on individual webpage runtime behaviors. We model these overheads in our performance evaluation and discuss their impact along with the improvements.

7.2 Style Resolution Unit

Our SRU prototype design achieves on average 3.5X, and up to 10X, speedup for the accelerated style applying phase. The improvements vary because of individual webpage characteristics.

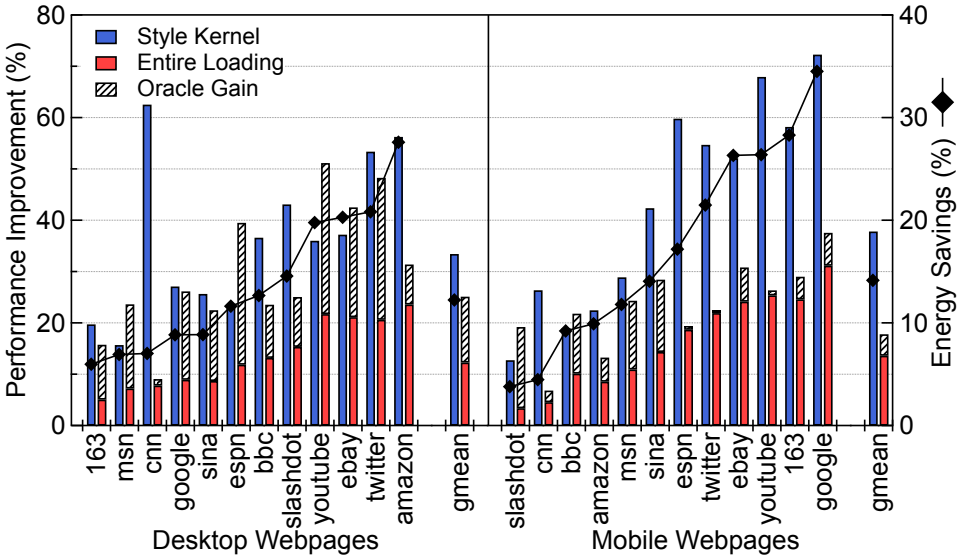


Fig. 16. Performance and energy improvement of the SRU.

Fig. 16 shows SRU's performance improvement for the *Style* kernel and the entire webpage loading on the performance-oriented design D2 in Fig. 5. The average performance improvement of the *Style* kernel is 33.4% and 37.8% for desktop and mobile webpages, respectively. Generally, we find that mobile webpages benefit slightly more from the SRU because they tend to be less diversified in webpage styling, and therefore the SRU has higher coverage.

The overall improvements vary across webpages because different webpages spend different portions of time in the *Style* kernel. For example, cnn spends only 14% of its execution time in the *Style* kernel during the entire run. Therefore, its 62% improvement in the *Style* kernel translates to an overall improvement of only 7%. On average, the SRU improves the entire webpage load time by 13.1% on all the webpages.

The SRU not only improves performance but also reduces energy consumption. The right y-axis of Fig. 16 shows the energy saving for the entire webpage loading. Webpages are sorted according to the energy savings. On average, SRU results in 13.4% energy saving for all webpages.

Fig. 16 also shows the oracle improvement if the entire applying phase can be delegated to the SRU (i.e., no hardware resource constraints). Desktop webpages have much higher oracle gain than mobile webpages. The software fall-back mechanism is more frequently triggered in desktop-version webpages due to their diversity in styling webpages. This also implies the potential benefits of reconfiguring the SRU according to different webpages. An SRU that is customized for mobile webpages could potentially be much smaller.

We apply the SRU to different designs to show its general applicability. For loading an entire webpage, on a current mobile processor design (D3), the SRU improves performance by 10.0% and reduces energy consumption by 10.3%. On an energy-conscious design (D1), it improves performance by 8.4% and reduces energy consumption by 11.6%.

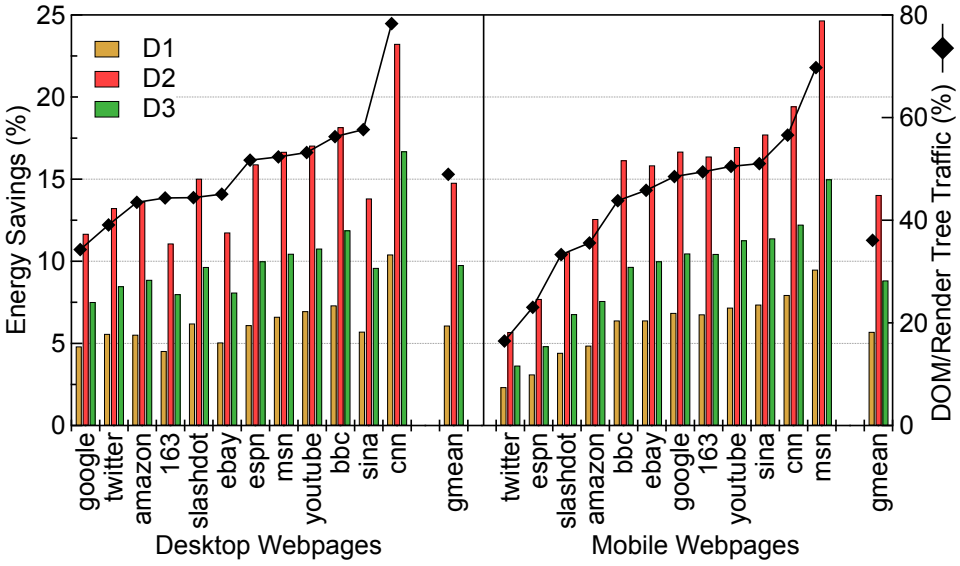


Fig. 17. Energy savings with a browser engine cache.

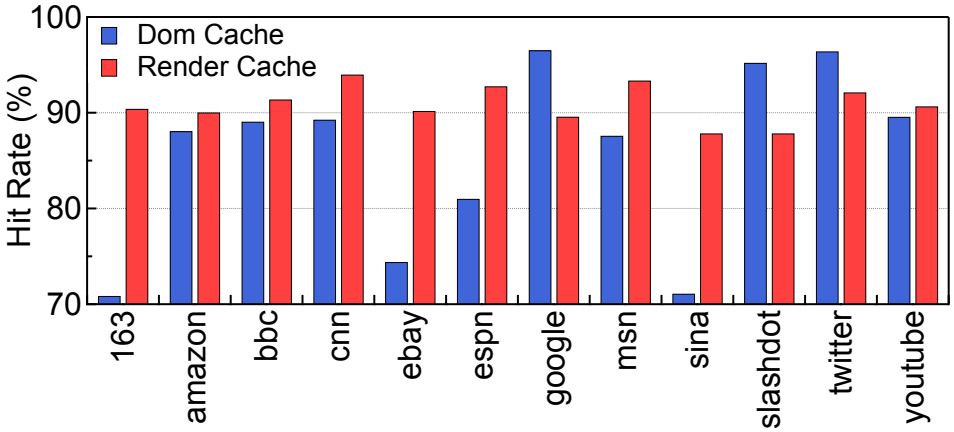


Fig. 18. DOM Cache and Render Cache hit rate for desktop webpages.

7.3 Browser Engine Cache

Fig. 17 shows the energy reduction from using the browser engine cache. The browser engine cache can serve data more energy-efficiently because of the high hit rate of its cache (as shown in Fig. 13b). Mobile webpages achieve less energy saving than desktop-version webpages because of their smaller memory footprint. On average, the performance-oriented design (D2) achieves 14.4% energy savings. Since the energy-conscious (D1) and current design (D3) have smaller caches, the energy consumption caused by the data cache is less, and therefore benefits less from the browser engine cache. On average, their energy consumption reduces by 5.9% and 9.3%, respectively.

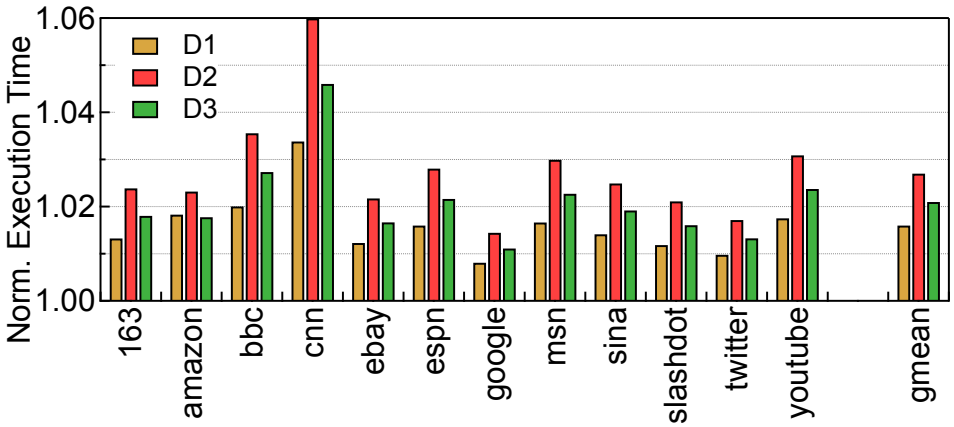


Fig. 19. Execution time with the browser engine cache of the three designs. Values are normalized to each design's baseline configuration without the browser engine cache.

We find that the DOM tree and Render tree access intensity largely determines the amount of energy saving. The right *y*-axis in Fig. 17 shows the amount of L1 data cache traffic that is attributed to accessing both data structures. In the most extreme case, about 80% of the data accesses for loading cnn touch the DOM tree and the Render tree. Therefore, it achieves the largest energy saving.

There are some outliers in desktop webpages where the energy savings are not proportional to DOM/Render tree access intensity. For example, `sina` has a much higher traffic (~60%) than `twitter` (~40%), but with similar energy savings. This is because `sina` has a much lower DOM cache hit rate than `twitter`. Fig. 18 shows the DOM cache and Render cache hit ratio for desktop webpages. We observe that `sina` has a DOM cache hit rate at ~70%, lower than `twitter` at ~97%. A lower DOM cache hit ratio indicates the `sina` does not fully use the low-energy browser engine cache. In contrast, we find that mobile webpages all have a high browser engine cache hit rate, and therefore their energy savings closely track the DOM/Render tree traffic.

Due to the software cache management overhead, the browser engine cache incurs performance overhead. Fig. 19 shows the desktop webpages' execution time of the three designs with the browser engine cache. The values are normalized to each design's baseline configuration without the browser engine cache. We find that the performance slow down is minimal, primarily because the design decisions that we made (as described in Sec. 6.2) minimize the software management overhead. On average, the slow down for D2 with a 64 KB L1 data cache is only 2.7%. The slow down for D1 and D3 with smaller L1 data caches (8 KB and 32 KB, respectively) is slightly smaller–only 1.6% and 2.1%, respectively. We speculate that the reason is that both D1 and D3 have slower performance than D2, and as such, they amortize the overhead of the software cache management.

7.4 Combined Evaluation

Fig. 20 shows the energy-efficiency improvement for the entire webpage loading on all three designs by progressively adding the two optimization techniques. The dotted curve represents the Pareto-optimal frontier of the design space discovered in Sec. 4.2. The circles represent original designs in this energy-performance space. The triangles represent the new energy-performance trade-off points after applying the software-managed browser engine cache optimization. The squares show the new energy-performance points when the SRU is added atop the caching optimization.

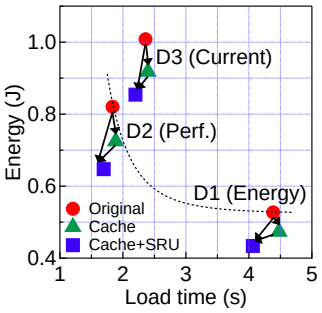


Fig. 20. Energy-efficiency improvement over three designs.

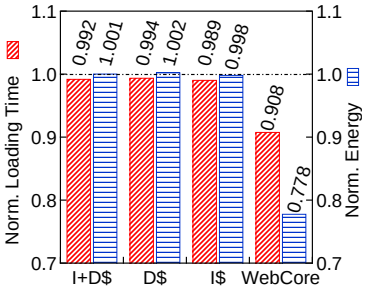


Fig. 21. Allocating area for caches versus specializations.

Comparing the energy-conscious design (D2) with an existing mobile processor design (D3), we observe that customization of the general-purpose architecture alone without applying any specialization allows us to achieve 22.2% performance improvement and 18.6% energy saving.

After applying the browser engine cache, the performance slightly degrades due to its software management overhead. Therefore, all the triangles move slightly to the right despite the energy savings. However, applying the SRU optimization improves both performance and energy consumption. All the squares move toward the left corner. In effect, we push the Pareto-optimal frontier in the original design space to a new design frontier with significantly better energy efficiency.

In addition, we also observe that D3 with our specializations can now approach the original Pareto-optimal frontier. This implies that it is possible to apply specializations to existing mobile processors to achieve a similar level of energy efficiency as processors that are optimized for the mobile Web browsing workloads.

On average, the energy-conscious design (D1) benefits by 6.9% and 16.6% for performance improvement and energy reduction, respectively. The performance-oriented design (D2) benefits by 9.2% and 22.2% for performance improvement and energy reduction, respectively. Lastly, the existing mobile processor design (D3) benefits by 8.1% and 18.4% for performance improvement and energy reduction, respectively.

Our specializations incur area overhead. To quantitatively assess the effectiveness of the area overhead, we compare our results with general-purpose designs that simply use the same area overhead to scale up microarchitecture resources. In our evaluation, we use the additional area to improve the I-cache and D-cache sizes because instruction delivery and data feeding are the two major bottlenecks, as discussed in Sec. 4.3. The additional area would be most justified to improve the I-cache and D-cache sizes.

As an example, Fig. 21 compares our combined specializations (WebCore) with designs that increase the I-cache size by 24 KB (I$), D-cache size by 24 KB (I$), and both caches by 12 KB (I+D$) based on the D2 design. The figure normalizes the webpage loading time and energy consumption to the D2 design without any specializations. We see that simply improving the cache sizes in general-purpose cores achieves only negligible performance improvement (¡1%) with a slightly higher energy consumption. However, WebCore specializations provide significantly better energy efficiency.

8 RELATED WORK

Web Browser Optimizations Prior software work focuses on parallelizing browser kernels/tasks for improving performance [8, 16, 53, 56–58, 60]. Although such parallelized algorithms can achieve speedups ranging from 4X to 80X for specific tasks, they typically do not scale well beyond four cores/threads. None of the mainstream Web browsers, such as Chrome and Firefox, explicitly parallelize browser rendering engine computations for

multiple cores. Multithreading is mostly only used for resource loading, such as network prefetching and TCP connections, rather than computation, which is the focus of this paper. Our measurement results on the Exynos 5410 SoC show that going from 2 to 4 cores doubles the power consumption while improving performance by only 10%. In addition, our optimizations target both performance and energy, and can therefore readily improve the per-thread/task energy efficiency.

Zhu and colleagues propose scheduling techniques that leverage the big/little heterogeneous system for optimizing the energy efficiency of mobile Web applications [86, 88, 91]. Big/little scheduling trade-off performance with energy consumption. In contrast, WebCore customization and specialization improve both performance and energy consumption at the same time. In addition, WebCore enriches the heterogeneity at the core and system level, thus creating more opportunity for the compiler and operating system scheduling.

GreenWeb [90] is a set of programming language support that allow developers to specify constraints about user QoS experience while delegating to the runtime system how to make calculated trade-offs between energy consumption and QoS experience. In comparison, WebCore is an architecture-level mechanism that does not rely on programming language support.

Similar to our work, SiChrome [11] performs aggressive specializations that map much of the Chrome browser into silicon, and achieves EDP improvement. The key difference is that we retain general-purpose programmability while still being energy efficient. Also, our Pareto-optimal analysis provides a more generic optimization view than the EDP-based evaluation.

ESP [18] and EFetch [17] also propose specialized hardware structures on top of general-purpose cores to improve the performance and energy efficient of Web applications. They view Web application execution has a sequence of events while we view Web application execution has a mix of different kernels. Both views are complementary in that per-event execution can benefit from kernel-level improvement that WebCore provides and vice versa.

Other works take a system-level perspective on improving Web browsing performance, such as asynchronous rendering, resource prefetching, and refactoring JavaScript and CSS files [52, 75, 79, 80, 85]. Our work is complementary to them beacuse they can all benefit from kernel-level efficiency improvements.

Web Applications Characterization BBench [29] is a webpage benchmark suite that includes 11 hot webpages. Its authors perform microarchitectural characterizations of webpage loading on an existing ARM system. Although the authors show that the 11 webpages have distinctly different characteristics from SPEC CPU 2006, they do not quantify the comprehensiveness and representativeness of the webpages against the vast number of webpages "in the wild." In stark contrast, our analysis in Sec. 3 systematically proves the broad coverage of our webpages, which is needed for robustly evaluating the impact of the optimizations that we propose. For example, we find that BBench does not include significantly complex webpages, and our analysis led to including two webpages of that sort, i.e., `www.163.com` and `www.sina.com.cn`. Their webpage sizes are about 4x larger than the average BBench webpage, and as such are needed to increase the coverage of our benchmarking suite.

MobileBench [63] characterizes the performance impact of various microarchitecture features on mobile workloads. Our paper quantifies the performance-energy trade-off, and focuses specifically on Web applications. Complementary to our design space exploration, MobileBench results show that more aggressive customizations of other microarchitecture structures such as the prefetcher are worth exploring.

JavaScript Our work is not about JavaScript execution. However, we found that a significant amount of JavaScript execution time is spent in the browser's kernels (~40%). Our work indirectly studies how the browser engine can improve JavaScript performance and energy efficiency. There are prior works on the JavaScript language engine itself, including analysis [69] and optimizations[27, 54, 55]. They are separate and complementary to our work involving the browser engine.

Specialization Alternatives L0 caches and scratchpad memories [9, 40] have long been used to reduce data communication overhead by acting as small, fast, and energy-conserving data storage. The browser engine cache

proposed in this paper demonstrates the effectiveness of such an idea for mobile Web browsing workloads. We propose to implement the browser engine cache as a collection of registers where each register holds exactly one DOM (render) tree attribute. In contrast, the typical L0 cache in mobile SoCs [41] is agnostic to the application-level data structures. Each L0 cache line, thus, holds more than one DOM attribute, leading to excessive energy consumption when accessing individual attributes.

In addition, the strong locality of the principal data structures revealed in our analysis can potentially be captured by dedicating cache ways to the Web browser application [23, 42]. The streaming access pattern of the DOM tree shown in Fig. 14 indicates that a dynamic cache insertion policy such as DIP [68] or an intelligent linked data structure prefetcher [22] on L1 data cache are also worth exploring. However, the browser engine cache we propose aims at saving energy with minimal loss in performance, which the prior performance-oriented techniques have not been proven/claimed to provide. Finally, hardware support for index walking and list traversal [43] could also potentially be useful for accelerating DOM tree walking.

9 CONCLUSION

The demise of graceful Dennard scaling, and consequently the dark silicon phenomenon, have been widely recognized in our community as an urgent challenge threatening the next generation of computing advancements. Improving energy efficiency is now becoming the first-class design consideration. Among all techniques, hardware customization and specialization is deemed an extremely effective approach to improve energy efficiency.

One of the most important questions to address as we enter the dark silicon era is to identify what application domains benefit the most from dedicated hardware resources and design efforts. In retrospect, software computational kernels that made their way into today's hardware have had a strong usage base and long-standing impact. Examples from recent proposals cover important domains such as scientific computing [67], video processing [32], and signal processing [82], but none specifically targets the Web application domain.

WebCore is the first attempt to comprehensively customize and specialize for the mobile application domain. Specifically, it identifies the Web software infrastructure (HTML, CSS, and JavaScript) as a promising target for hardware customization and specialization in the complex mobile software ecosystem. Customizations identify the general-purpose baseline architecture that uniquely matches the Web workload's needs. Specializations further pack enough domain-specific computations (SRU) and support energy-efficient data communication across kernels (browser engine cache). Altogether, they push the energy-efficiency frontier of general-purpose mobile processor designs to a new level for mobile Web browsing workloads. Such designs are warranted given current mobile processor architecture trends in a battery-constrained energy envelope.

Longevity As we see it, the longevity of the WebCore lies in the following aspects. First, the Web platform is, and will continue to be, the substrate of many Web applications due to its "write-once, run-anywhere" feature that tackles the notorious device fragmentation issue [72]. Exemplifying this design philosophy is Google's Portable Native Client (PNaCl). It supports the porting of native C/C++ applications to the Chrome browser [24]. SkyFire Technology shows that Web applications based on browser technologies still far outweigh native apps, excluding games [25]. Even for gaming, we see a burst of advanced browser-based games owing to the emergence and widespread adoption of HTML5 technologies. New gaming libraries such as Construct2 [19] make it possible to port entire real-time physics engines such as the Unreal Engine into a browser [46].

Principled Approach WebCore is not specific to a particular browser implementation; rather, it targets important computation and communication patterns that are fundamental to any Web browser. Those patterns are generally found across different Web browser engines, such as WebKit and Gecko. In addition, the kernel algorithms and data structures remain largely unchanged across browser versions. For example, the algorithm we study in the *Style* kernel has remained almost identical over the past two years, which includes over 10 versions of Chromium. Therefore, we do not expect software changes to dramatically impact our hardware design.

Balancing Programmability and Specialization Unlike prior research that takes either a fully software approach on general-purpose processors [16, 56] or a fully hardware approach [11], WebCore strikes a balance between the two. On one hand, WebCore retains the flexibility and programmability of a general-purpose core. The general-purpose programmability is essential to support the complex browser software system, and allows fast prototyping of new ideas and implementations. On the other hand, it incorporates modest hardware specializations that create closely coupled datapath and data storage to achieve energy-efficiency improvements.

Heterogeneous Architecture Our vision of the WebCore is that it is one core of a heterogeneous multicore SoC, tuned specifically for Web workloads. Normal workloads can use regular cores. Different from the typical big/little type of heterogeneous processors, WebCore increases the system heterogeneity by providing domain-specific hardware specialization. Recent industry efforts, such as hardware support for WebRTC in Tegra 4 [61], reinforce this emerging new trend. WebCore can be integrated with other heterogeneous proposals that improve Web browsing efficiency (such as via scheduling [88]).

REFERENCES

[1] 7-cpu. 2017. ARM Cortex-A15 Specification. http://goo.gl/CXYook. (2017).
[2] Alexa. 2017. Alexa. http://www.alexa.com/. (2017).
[3] ARM. 2011. Enabling Mobile Innovation with the Cortex-A7 Processor. http://www.arm.com/about/events/enabling-mobile-innovation-with-the-cortex-a7-processor.php. (2011).
[4] ARM. 2012. Exploring the Design of the Cortex-A15 Processor. http://goo.gl/Pc8hPe. (2012).
[5] ARM. 2015. ARM Cortex A15. http://www.arm.com/products/processors/cortex-a/cortex-a15.php. (2015).
[6] ARM. 2015. ARM DS-5. http://ds.arm.com/ds-5/optimize/. (2015).
[7] O. Azizi, A. Mahesri, B.C. Lee, S. J. Patel, and M. Horowitz. 2010. Energy Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In *Proc. of ISCA*.
[8] Carmen Badea, Mohammad R. Haghighat, Alexandru Nicolau, and Alexander V. Veidenbaum. 2010. Towards Parallelizing the Layout Engine of Firefox. In *Proc. of USENIX HotPar*.
[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. 2002. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proc. of CODES+ISSS*.
[10] Battery University. 2011. Battery Statistics. http://goo.gl/90mMeb. (2011).
[11] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steven Swanson, and Michael Bedford Taylor. 2012. SiChrome: Mobile Web Browsing in Hardware to Save Energy. *DaSi: First Dark Silicon Workshop* (2012).
[12] Joshua Bixby. 2011. 2012 Predictions: The Average Web Page Will Hit 1 MB, Google and Siri Will Face Off, and Chrome, Windows 7, and RUM will rise. http://goo.gl/WmcTsx. (2011).
[13] Joshua Bixby. 2011. The Relationship Between Faster Mobile Sites and Business KPIs: Case Studies from the Mobile Frontier. http://goo.gl/shnlDF. (2011).
[14] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *Proc. of HPCA*.
[15] Michael Butler, Tse-Yu Yeh, Yalt Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. 1991. Single Instruction Stream Parallelism Is Greater than Two. In *Proc. of ISCA*.
[16] Calin Cascaval, Seth Fowler, Pablo Montesinos Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. 2013. Zoomm: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of PPoPP*.
[17] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2014. EFetch: Optimizing Instruction Fetch for Event-driven Web Applications. In *Proc. of PACT*.
[18] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2015. Accelerating Asynchronous Programs through Event Sneak Peek. In *Proc. of ISCA*.
[19] Construct2. 2015. Construct2. https://www.scirra.com/construct2. (2015).
[20] G. Dunteman. 1989. *Principal Component Analysis.* Sage Publications.
[21] Kit Eaton. 2013. How 1s Could Cost Amazon $1.6 Billion in Sales. http://goo.gl/qG0M2Q. (2013).
[22] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proc. of HPCA*.
[23] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. 2011. Buffer-Integrated-Cache: A Cost-effective SRAM Architecture for Handheld and Embedded Platforms. In *Proc. of DAC*.
[24] Gigaom. 2013. Portable Native Client. http://goo.gl/Olm3NP. (2013).

[25] Jeffrey Glueck. 2011. Why Flurry Got It Wrong On Mobile Apps Vs. Web Browsers. http://www.businessinsider.com/why-flurry-got-it-wrong-on-apps-v-browsers-2011-6. (2011).
[26] Google. 2015. Chromium browser. http://www.chromium.org/Home. (2015).
[27] Lauren Guckert, Mike O'Connor, Satheesh Kumar Ravindranath, Zhuoran Zhao, and Vijay Janapa Reddi. 2013. A Case for Persistent Caching of Compiled JavaScript Code in Mobile Web Browsers. In *Workshop on AMAS-BT*.
[28] Qi Guo, Tianshi Chen, Yunji Chen, Zhihua Zhou, Weiwu Hu, and Zhiwei Xu. 2011. Effective and Efficient Microprocessor Design Space Exploration Using Unlabeled Design Configurations. In *Proc. of IJCAI*.
[29] A. Gutierrez, R. Dreslinski, A. Saidi, C. Emmons, N. Paver, T. Wenisch, and T. Mudge. 2011. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *Proc. of IISWC*.
[30] Erik G. Hallnor and Steven K. Reinhardt. 2000. A Fully Associative Software-managed Cache Design. In *Proc. of ISCA*.
[31] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. 2016. Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction. In *Proc. of HPCA*.
[32] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proc. of ISCA*.
[33] Hardkernel. 2015. ODROID-XU+E Development Board. http://goo.gl/Ige0Jp. (2015).
[34] Frank E Harrell. 2001. *Regression Modeling Strategies*. Springer.
[35] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning*. Springer.
[36] Urs Hoelzle. 2012. The Google Gospel of Speed. https://goo.gl/fTd0f0. (2012).
[37] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. of MobiSys*.
[38] Intel. 2012. Intel Atom Processor Z2460. http://download.intel.com/newsroom/kits/ces/2012/pdfs/AtomprocessorZ2460.pdf. (2012).
[39] Intel. 2013. Technology Insight: Intel Silvermont Microarchitecture. http://www.anandtech.com/show/6936/intels-silvermont-architecture-revealed-getting-serious-about-mobile. (2013).
[40] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. 1997. The Filter Cache: an Energy Efficient Memory Structure. In *Proc. of MICRO*.
[41] Brian Klug and Anand Lal Shimpi. 2011. Krait Cache and Memory Hierarchy. http://goo.gl/ZuO7X2. (2011).
[42] Theo Kluter, Philip Brisk, Edoardo Charbon, and Paolo Ienne. 2013. Way Stealing: A Unified Data Cache and Architecturally Visible Storage for Instruction Set Extensions. In *IEEE Transactions on VLSI*.
[43] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-Memory Databases. In *Proc. of MICRO*.
[44] Kssmetrics. 2011. How Loading Time Affects Your Bottom Line. https://goo.gl/XlWGBK. (2011).
[45] Kssmetrics. 2011. Speed is a Killer. http://goo.gl/4PfsJL. (2011).
[46] Frederic Lardinois. 2013. Mozilla And Epic Games Bring Unreal Engine 3 To The Web. http://techcrunch.com/2013/03/27/mozilla-and-epic-games-bring-unreal-engine-3-to-the-web-no-plugin-needed/. (2013).
[47] Benjamin C. Lee and David M. Brooks. 2006. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proc. of ASPLOS*.
[48] Paul Lewis. 2014. Rendering Performance. https://goo.gl/Ff5HrD. (2014).
[49] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of MICRO*.
[50] Yingmin Li, Mark Hempstead, Patrick Mauro, David Brooks, Zhigang Hu, and Kevin Skadron. 2005. Power and Thermal Effects of SRAM vs. Latch Mux Design Styles and Clocking Gating Choices. In *Proc. of ISLPED*.
[51] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2006. SODA: A Low-power Architecture For Software Radio. In *Proc. of ISCA*.
[52] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. 2012. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proc. of ASPLOS*.
[53] Haohui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Montesinos Pablo. 2012. A Case for Parallelizing Web Pages. In *Proc. of USENIX HotPar*.
[54] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. 2011. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In *Proc. of HPCA*.
[55] Mojtaba Mehrara and Scott Mahlke. 2011. Dynamically Accelerating Client-side Web Applications Through Decoupled Execution. In *Proc. of CGO*.
[56] Leo A. Meyerovich and Rastislav. Bodik. 2010. Fast and Parallel Webpage Layout. In *Proc. of WWW*.
[57] Leo A. Meyerovich and Rastislav Bodik. 2012. FTL: Synthesizing a Parallel Layout Engine. In *Proc. of ECOOP*.
[58] Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proc. of PPoPP*.

[59] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. 2012. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Proc. of MICRO*.
[60] Mozilla. 2015. Servo. https://github.com/mozilla/servo. (2015).
[61] NVidia. 2013. Hardware Support for WebRTC in Tegra4. http://blogs.nvidia.com/blog/2013/05/17/nvidia-shows-off-first-1080p-high-def-mobile-video-conferencing-at-google-io-with-tegra-4/. (2013).
[62] OpenHub. 2017. Chromium Project Summary: Languages. https://goo.gl/XQb3EO. (2017).
[63] Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu. 2013. Performance, Energy Characterizations and Architectural Implications of an Emerging Mobile Platform Benchmark Suite-MobileBench. In *Proc. of IISWC*.
[64] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proc. of DAC*.
[65] Yale N Patt, Sanjay J Patel, Marius Evers, Daniel H Friendly, and Jared Stark. 1997. One Billion Transistors, One Uniprocessor, One Chip. *Computer* 30, 9 (1997), 51–57.
[66] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy John. 2006. *Four Generations of SPEC CPU Benchmarks: What Has Changed and What Has Not.* Technical Report LCA-TR-041026-01-1. The University of Texas as Austin.
[67] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. 2013. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. In *Proc. of ISCA*.
[68] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely., and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proc. of ISCA*.
[69] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. 2009. JSMeter: Characterizing Real-world Behavior of JavaScript Programs.
[70] Samsung. 2015. Samsung Exynos 5410 SoC. http://goo.gl/KpbHm3. (2015).
[71] Fred Schlachter. 2013. No Moore's Law for Batteries. In *Proc. of National Academy of Science of the United States of America*.
[72] Open Signal. 2014. Android Fragmentation Visualized. http://goo.gl/ODlx4z. (2014).
[73] Shikhir Singh. 2015. HTML5 On The Rise: No Longer Ahead Of Its Time. http://goo.gl/yuEVCy. (2015).
[74] Mac Slocum. 2011. You Can't Get Away With a Bad Mobile Experience Anymore. http://goo.gl/T3812z. (2011).
[75] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. 2012. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proc. of WWW*.
[76] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1.
[77] W3C. 2014. CSS Cascading Order. https://goo.gl/PkKg92. (2014).
[78] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A memory-system simulator. In *Computer Architecture News*.
[79] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why are Web Browsers Slow on Smartphones?. In *Proc. of HotMobile*.
[80] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How Far Can Client-Only Solutions Go for Mobile Browser Speed?. In *Proc. of WWW*.
[81] WebKit. 2015. WebKit. http://www.webkit.org. (2015).
[82] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2009. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of ISCA*.
[83] Dong Ye, Joydeep Ray, Christophe Harle, and David Kaeli. 2006. Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture. In *Prof. of IISWC*.
[84] Allan Yogasingam. 2013. Teardown: Samsung Galaxy S4. http://goo.gl/BQl4Dg. (2013).
[85] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. 2010. Smart Caching for Web Browsers. In *Proc. of WWW*.
[86] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. 2015. Event-based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications. In *Proc. of HPCA*.
[87] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. 2015. The Role of the CPU in Energy-Efficient Mobile Web Browsing. In *Micro, IEEE*.
[88] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. In *Proc. of HPCA*.
[89] Yuhao Zhu and Vijay Janapa Reddi. 2014. WebCore: Architectural Support for Mobile Web Browsing. In *Proc. of ISCA*.
[90] Yuhao Zhu and Vijay Janapa Reddi. 2016. GreenWeb: Language Extensions for QoS-aware Energy-Efficient Mobile Web Computing. In *Proc. of PLDI*.
[91] Yuhao Zhu, Aditya Srikanth, Jingwen Leng, and Vijay Janapa Reddi. 2014. Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing. In *Computer Architecture Letters*.

Received February 2007; revised March 2009; accepted June 2009