WebCore: Architectural Support for Mobile Web Browsing

Yuhao Zhu Vijay Janapa Reddi
Department of Electrical and Computer Engineering
The University of Texas at Austin
yzhu@utexas.edu, vj@ece.utexas.edu

Abstract

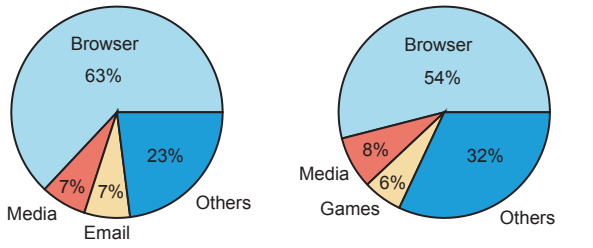
The Web browser is undoubtedly the single most important application in the mobile ecosystem. An average user spends 72 minutes each day using the mobile Web browser. Web browser internal engines (e.g., WebKit) are also growing in importance because they provide a common substrate for developing various mobile Web applications. In a user-driven, interactive, and latency-sensitive environment, the browser's performance is crucial. However, the battery-constrained nature of mobile devices limits the performance that we can deliver for mobile Web browsing. As traditional general-purpose techniques to improve performance and energy efficiency fall short, we must employ domain-specific knowledge while still maintaining general-purpose flexibility.

In this paper, we first perform design-space exploration to identify appropriate general-purpose architectures that uniquely fit the characteristics of a popular Web browsing engine. Despite our best effort, we discover sources of energy inefficiency in these 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 energy efficiency and at the same time improve mobile Web browsing performance. As emerging mobile workloads increasingly rely more on Web browser technologies, the type of optimizations we propose will become important in the future and are likely to have lasting widespread impact.

1. Introduction

The proliferation of mobile devices and the fast penetration of new Web technologies (such as HTML5) has ushered in a new era of mobile Web browsing. Mobile users now spend a significant amount of time on the Web browser every day. A study conducted by our industry partner reports that the browser occupies 63% of the window focus time on their employees' mobile devices, as shown in Fig. 1a. As a general trend, comScore shows that mobile users prefer Web browsers over native applications for important application domains such as online e-commerce and electronic news feeds [1].

In the user-centric mobile Web browsing context, delivering high performance is critical to end users' quality-of-service experience. Studies in 2013 indicate that 71% of mobile Web users expect website performance on their mobile devices to be not worse than their desktop experience–up from 58% in 2009 [5]. Traditionally, Web browsing performance has been network limited. However, this trend is changing. With about 10X improvement in round-trip time from 3G to LTE, network latency is no longer the *only* performance bottleneck [51]. 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 requirements have increased by as much as 10X [78]. Fig. 1b confirms that the browser consumes a significant portion of the CPU time. Similarly, other research reports over 80% CPU usage for mobile browsing [51].



(a) Time dist. of window focus. (b) Time dist. of CPU processing.

Fig. 1: Mobile Web browser share study conducted by our industry research partner on their employees' devices [2]. Similar observations were reported by NVIDIA on Tegra-based mobile handsets [3,4].

Providing high processing capability in mobile CPUs is challenging due to the limited battery capacity. Recent studies suggested that the advancement in Lithium-ion battery density has slowed down significantly, and that battery capacity will be mostly limited by its volume [6, 72]. Under such a constraint, we are now faced with a challenge. On one hand, the energy budget for mobile devices is unlikely to drastically increase in the short term. On the other hand, mobile processors are becoming power hungry. Core designs have not only gone from in-order to out-of-order (e.g., ARM Cortex-A8 to A15), but they have also gone multicore (e.g., Exynos 5410 in Samsung Galaxy S4 has eight cores). Experiments conducted by Carroll et al. suggested that processor power of mobile Web browsing doubled from 2010 to 2013 [38, 39].

Our work aspires to bridge the widening gap between high-performance and energy-constrained mobile processor designs for Web browsing. Domain-specific specializations have long been known to be extremely energy efficient [57, 71, 76]. Recent proposals in data computation domains (such as H.264 encoding [49] and convolution [67]) have begun showing that it is critical and feasible to balance the efficiency of application-specific specializations with general-purpose programmability.

978-1-4799-4394-4/14/$31.00

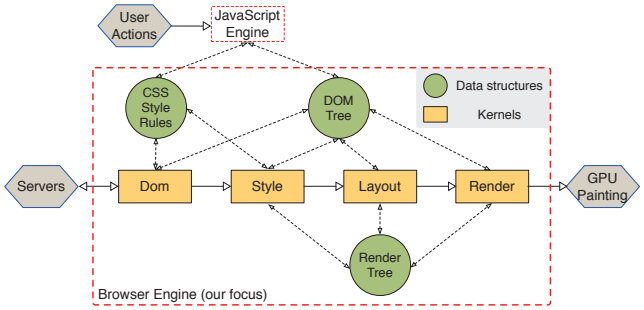


Fig. 2: Web browser overview.

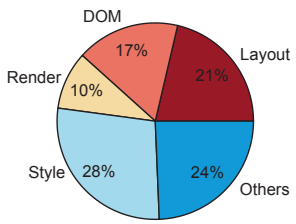


Fig. 3: Execution time breakdown of the browser's kernels.

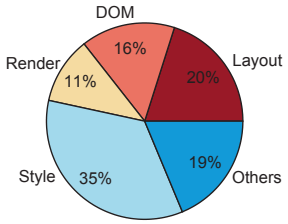


Fig. 4: Energy consumption breakdown of the kernels.

Sharing the same architecture design philosophy, we propose the WebCore, *a general-purpose core customized and specialized for the mobile Web browsing workload*. In comparison to prior work that either takes a fully software approach on general-purpose processors [41, 62] or a fully hardware specialization approach [34], 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 browsing workload. 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 them by tuning key design parameters achieves better energy efficiency (Sec. 4).

Building on the customized general-purpose baseline, we develop specialized hardware to further overcome the instruction delivery and data feeding bottlenecks (Sec. 5). We propose two new optimizations: the *"Style Resolution Unit"* (SRU) and a *"Software-Managed Browser Engine Cache."* The SRU is a software-assisted hardware accelerator for the critical style-resolution kernel within the Web browser engine. It exploits fine-grained parallelism that aggregates enough computation to offset the instruction and data communication overhead. The proposed cache structure exploits the unique data locality of the browser engine's principal data structures. It is a small and fast memory that 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.

2. Web Browser Background and Overview

We first describe the Web browser engine's computation kernels and their communication patterns. Such understanding helps us design effective customizations and specializations, such as those described in the later sections. Nearly all mainstream browser engines fit into our description. In addition, we show each kernel's performance and energy breakdown to show their importance and demonstrate our study's coverage.

Browser engine kernels Fig. 2 shows the overall flow of execution within any typical Web browser. The engine has two core modules: the browser engine (e.g., WebKit for Chrome and Gecko for Firefox) and the JavaScript engine. In this work, we focus on the internals of the Web browser engine. The JavaScript engine's performance 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.

The browser engine mainly consists of four kernels: *Dom*, *Style*, *Layout*, and *Render*. The kernels, shown in boxes, process the webpage and prepare pixels for a GPU to paint. The figure 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 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 $< x, y >$ 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.

Performance Fig. 3 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 [7] while navigating the benchmarked web-

pages described in Sec. 3 using Chromium [8]. On average, the kernels consume 75% percent of the total execution time. The *Style* kernel is the most time-consuming task. Please refer to Sec. 7 for discussion on multicores for Web browsing.

Energy Fig. 4 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. 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 engine We focus on the popular WebKit [9] browser 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.

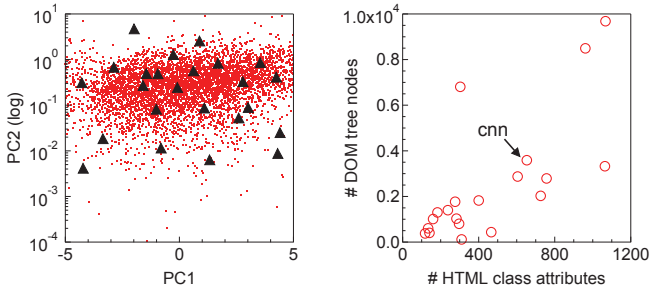
Benchmarked webpages We pay close attention to the choice of webpages to ensure that the WebCore design is not misled. We mine through the top 10,000 websites as ranked by Alexa [10] and pick the 12 most representative websites. All except one happen to rank among Alexa's top 25 websites.

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 [11, 12]. Moreover, many mobile devices, especially tablets, typically load the desktop version of webpages by default. As webpage sizes exceed 1 MB [13], we must study mobile processor architectures that can process more complex content and not just simple mobile webpages.

We study 24 distinct webpages. 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 [42]. 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. 5a 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.

Load time Unless stated otherwise, we define webpage load time as the amount of execution time that elapses until the `onload` event is triggered by the Web browser.

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 [35]. Therefore, we believe that our microarchitecture explorations are generally valid across ISAs. We use Marss86 [66], 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 [55] for power estimation.



(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. 5: Benchmark representativeness analysis.

4. Customizing the General-Purpose Cores

The industry has built both in-order (such as ARM Cortex A7 [14] and Intel Saltwell [15]) and out-of-order (such as ARM Cortex A15 [16] and Intel Silvermont [17]) 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 bear 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).

4.1. The 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 (e.g., issue width and the number of functional units) from reaching a completely unbalanced design [37]. In our study, 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, since thermal design power (TDP) is important for mobile SoCs, we eliminate overly aggressive designs with more than 2 W TDP.

We assume a fixed core frequency in our design-space exploration. We use 1.6 GHz, a common value in mobile processors [18, 19], to further prune the exploration space. However, because the latency of both the L1 and L2 caches can still vary, we include different cache designs in the exploration space.

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

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 [20], 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.

Since we consider billions of design points, it is not feasible to simulate all of them simply due to time constraints. Therefore, we leverage regression modeling techniques [50] to predict the performance and power consumption of various design points in the space. Such effort has been used successfully in the past for architecture design-space exploration [46, 54].

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. 5b 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 [78]. 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 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 construct the performance and power models for the four kernels described in Sec. 2, as well as the entire Web browser engine. In general, the out-of-order models' error rates are below 6.0%. The in-order models are more accurate because of their simpler design. On average, the in-order performance and power models' errors are within 5% and 2%, respectively.

Table 2: Microarchitecture configurations for the selected design points in Fig. 6 that represent the different energy-delay trade-offs.

	P1:OoO	P1:In-O	P2
Issue width	1	2	3
# Functional units	2	2	3
Load queue size (# entries)	4	N/A	16
Store queue size (# entries)	4	N/A	16
Branch prediction size (# entries)	1024	1024	128
ROB size (# entries)	128	N/A	128
# Physical registers	128	N/A	140
L1 I-cache size (KB)	64	128	128
L1 I-cache delay (cycles)	1	2	2
L1 D-cache size (KB)	8	64	64
L1 D-cache delay (cycles)	1	1	1
L2 cache size (KB)	256	1024	1024
L2 cache delay (cycles)	16	16	16

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 browser engine Fig. 6 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. To clearly illustrate the energy-performance trade-offs, we show only the Pareto-optimal design frontiers. Design points on the frontier reflect different optimal design decisions given specific performance/energy goals. 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 [31].

We make two important observations from Fig. 6. First, the out-of-order design space offers a much larger performance range ($\sim$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 has a low marginal performance value (i.e., high marginal cost of energy).

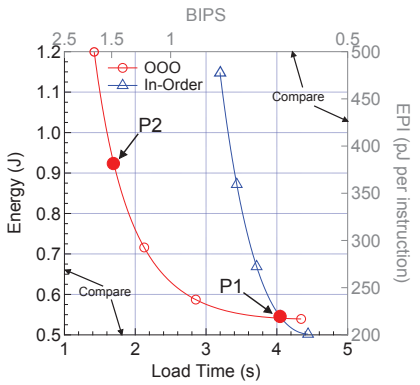


Fig. 6: In-order vs. out-of-order designs.

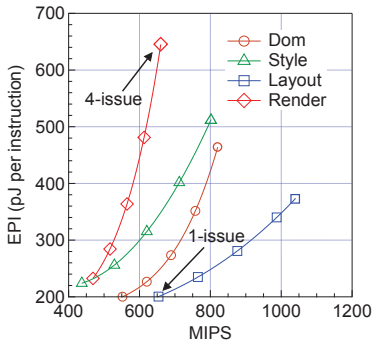


Fig. 7: Per-kernel in-order designs.

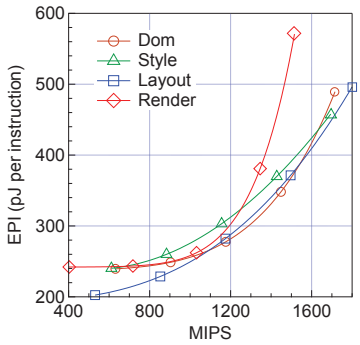


Fig. 8: Per-kernel out-of-order designs.

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. 6. Table 2 lists both configurations. 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 in-order execution logic that inhibits better performance than its out-of-order counterpart.

Going beyond the P1 crossover point (i.e., < 4 seconds), 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. We do not show the data due to space constraints.

Individual kernels To further understand why the in-order design is unsuitable for Web browser workloads, we study the individual kernels' behavior. Fig. 7 and Fig. 8 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.

However, such pronounced kernel variance is not present in out-of-order designs. For example, at 1200 MIPS, which is the "*knee of the curve(s)*," all the kernels have similar EPIs. Upon inspection, we find that all kernels have similar microarchitecture structures. 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), indicating that the out-of-order engine can explore the ILP for each kernel without bias. We do not show the data due to space constraints.

4.3. Energy Inefficiency in the Customized Core Designs

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 examine two specific optimization points (i.e., P1 and P2) in Fig. 6 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.

We find that the P1 and P2 configurations are different from current mobile processor designs. Table 2 summarizes the microarchitecture parameters optimized under the two optimization goals. 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. The performance-oriented design P2 also requires a larger data cache. In addition, the L2 cache in current mobile SoCs is typically 1 MB [16, 17], which accommodates all the applications on the core. However, the Web browsing workload alone requires a 1 MB L2 cache at P2. Let us explain our findings.

Instruction delivery Delivering instructions for execution is a major issue in the P1 and P2 designs. Both designs require a large L1 instruction cache and a large number of physical registers to alleviate the pressure on instruction fetching and dispatching. For instance, our results show that a 128 KB instruction cache reduces MPKI by 75% compared with an 8 KB cache. Although a larger L1 cache is more expensive to access, it reduces a significant amount of L2 cache accesses. In effect, it increases the L2 cache size. The insight here is useful in avoiding the excessive cost of the large L2 cache accesses in terms of static and dynamic power consumption.

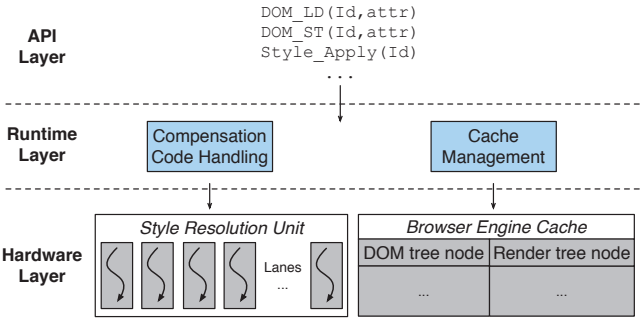


Fig. 9: Hardware-software co-design framework.

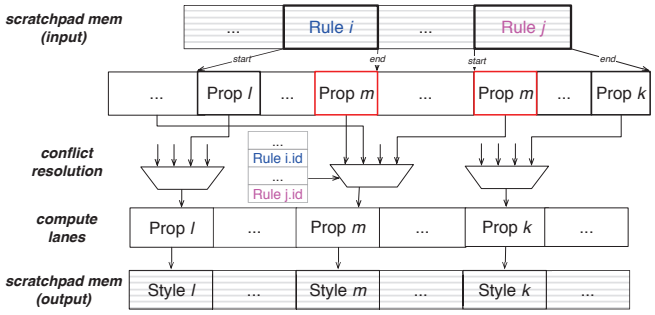


Fig. 10: SRU coupled with scratchpad memories.

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. Optimizing for P2 in Fig. 6 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.

5. Specializing the Customized Cores

Unusual design parameters in a processor core tuned for the mobile Web browsing workload indicates that both instruction delivery and data feeding are critical to guarantee high performance while still being energy efficient. In this section, we propose hardware and software collaborative mechanisms that mitigate the instruction delivery and data feeding inefficiencies in the customized out-of-order core designs. We introduce two new hardware enhancements, called the Style Resolution Unit (Sec. 5.1) and the Browser Engine Cache (Sec. 5.2). These new hardware structures are accessed via a set of high-level language APIs implemented as a runtime library (Sec. 5.3). As the Fig. 9 shows, the hardware supports fast and energy-efficient execution and data communication, and the library manages the hardware layer and thus eases the development effort for the WebCore. We first focus on the hardware design and then describe the runtime support.

5.1. Style Resolution Unit

The *Style* kernel takes about one-third of the execution time and the energy consumption of webpage loading as shown in Sec. 2. Therefore, optimizing the *Style* kernel would improve the overall energy efficiency the most. 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 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.

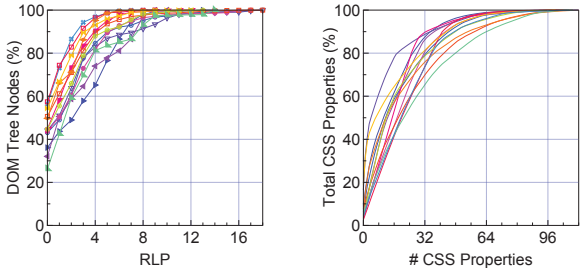
Overview The *Style* kernel consists of two phases: a matching phase and an applying phase. Previous work [41,62] focuses on parallelizing the matching phase. However, in our profiling, we find that the applying phase takes nearly twice as long to execute as the matching phase. Therefore, we focus on the applying phase. The applying phase takes in a set of CSS rules as input, applies each rule in the correct cascading order [21] to calculate each style property's final value (e.g., the exact-color RGB values, width pixels). The final values are stored back to the render tree (Fig. 2).

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.

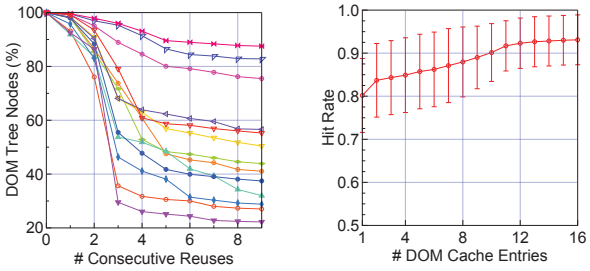
PLP follows RLP. Each rule has multiple properties, and each property is examined by the engine to set the corresponding data field in the render tree according to its property ID. Because properties are independent of one another, handling of their processing routines can be dealt with in parallel.

Proposed 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.



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

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



(a) DOM node reuse behavior. (b) DOM node access hit rate.

Fig. 12: DOM tree access behavior across webpages.

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 [78] 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.

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.

5.2. Software-Managed Browser Engine Cache

The DOM tree and Render tree are the two most important data structures because they are shared across different kernels, as shown in Fig. 2. We propose the *Browser Engine Cache* to improve the energy-efficiency of accessing these data structures. In Sec. 8, we discuss specializations for performance.

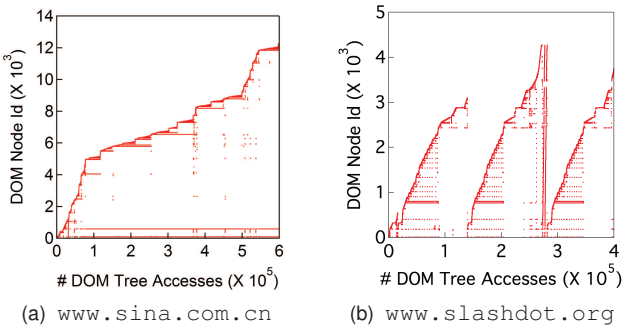
Our cache design is motivated by the unique access patterns to the DOM tree and render tree. They have strong locality that can benefit from a small and energy-efficient L0 cache memory, rather than the large power-hungry traditional caches.

The problem of the traditional cache is best embodied in the performance-oriented design P2 in Table 2. 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.

Overview The browser engine cache consists of a DOM cache and a Render cache. We use the DOM to explain our locality observation. Similar analysis and design principles also apply to the render cache. Fig. 12a 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 three times, which reflects strong data locality. This indicates that a very small cache can probably achieve the same hit rate as a regular cache, but with much lower power.

Such strong data reuse is due to intensive DOM tree traversals in the rendering engine. To illustrate this, Fig. 13 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. The browser engine typically operates on one DOM tree node heavily and traverses to the next one. Many kernels require such traversals. 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.

Proposed 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. Because each node has multiple attributes that must be individually accessed, we implement



(a) www.sina.com.cn (b) www.slashdot.org

Fig. 13: Representative DOM tree access patterns.

each cache line as a set of independently addressable registers, where each register holds one attribute of the DOM tree node.

Design considerations It is possible to implement the DOM cache entirely in hardware, similar to a normal data cache. However, we choose to implement it 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, as we will discuss in Sec. 5.3. Prior work has demonstrated effective software-managed cache implementations [48].

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. Moreover, the software overhead 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. 13, 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. 12b 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 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.

5.3. Software Support and Programmability

The SRU and browser engine cache 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 the need for being aware of the existence of the specialized hardware. It is important to note that unlike conventional programming models for accelerators, where the task invocation and completion semantics are tightly coupled [40], WebCore APIs can be freely mixed in with high-level programming.

SRU Programmers issue `Style_Apply(Id)` to trigger the style resolution task. Since not all the CSS properties are implemented in the SRU (as discussed in Sec. 5.1), the runtime library must first examine all the input properties. For properties that can be offloaded to the SRU, the library 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.

Browser Engine Cache Application developers issue `DOM_LD(Id, attr)` and `DOM_ST(Id, attr)` to access a particular attribute of a particular DOM tree node. The runtime library performs the actual hardware memory accesses as well as cache management, such as replacement and insertion. For example, the runtime 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 implements a cache simulator. However, the runtime overhead is negligible due to the simple cache design (as described in Sec. 5.2).

6. WebCore Evaluation

In this section, we first present the power and timing overhead analysis of the optimizations (Sec. 6.1). We then evaluate the energy-efficiency implications of the SRU and the browser engine cache individually (Sec. 6.2, Sec. 6.3). In the end, we show the energy-efficiency improvement combining both optimizations (Sec. 6.4). 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. 6. Similarly, D2 refers to the performance-oriented design (P2) in Fig. 6. 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, 140 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.

6.1. Overhead Analysis

We use CACTI v5.3 [22] 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 prop-

erty'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 [23]).

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 [56]. 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.

6.2. 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. Due to the space constraints, in general, we mostly focus on the overall browser-level workload improvements.

Fig. 14 shows SRU's performance improvement for the *Style* kernel and the entire webpage loading on the performance-oriented design D2 in Fig. 6. 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.

Because different webpages spend different portions of time in the *Style* kernel, the overall improvements vary across webpages. 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. 14 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. 14 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

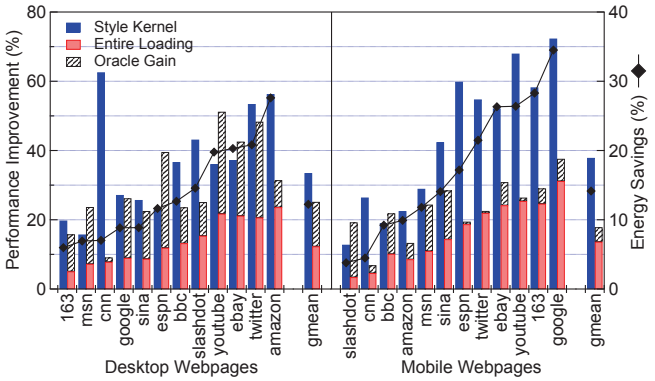


Fig. 14: Performance and energy improvement of the SRU.

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%.

6.3. Browser Engine Cache

Fig. 15 shows the energy reduction from using the browser engine cache. 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.

We find that the DOM tree and render tree access intensity largely determines the amount of energy saving. The right *y*-axis in Fig. 15 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. 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 (∼70%) than `twitter` (∼97%), and therefore does not fully use the low-energy browser engine cache. In contrast, mobile webpages have more regular access patterns. They 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. However,

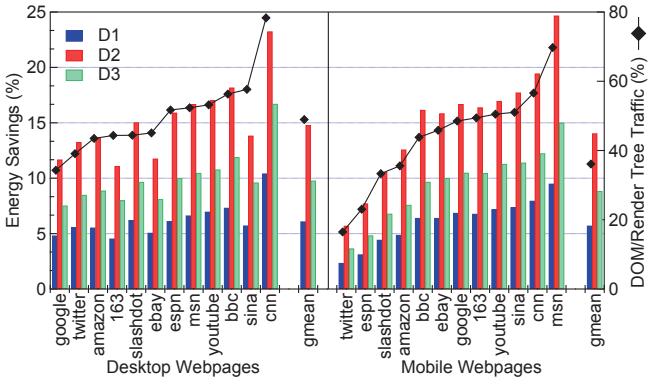


Fig. 15: Energy savings with a browser engine cache.

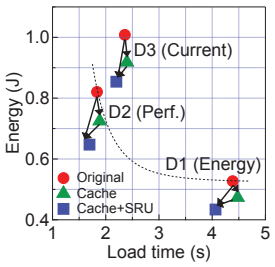


Fig. 16: Energy-efficiency improvement over three designs.

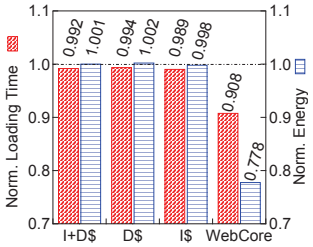


Fig. 17: Allocating area for caches versus specializations.

the design decisions that we made (as described in Sec. 5.2) minimize the software management overhead; on average, the slowdown for D2 with a 64 KB L1 data cache is only 2.5%.

The slowdown 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.

6.4. Combined Evaluation

Fig. 16 shows the energy-efficiency improvement for the entire webpage loading on all three designs by progressively adding the two optimization techniques. The dotted curve is the Pareto-optimal frontier of the design space discovered in Sec. 4.2. 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.

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.

Because our specializations incur area overhead, we also compare our results with designs that use the same area overhead to improve structures in general-purpose cores. Since instruction delivery and data feeding are the two major bottlenecks, as discussed in Sec. 4.3, the additional area would be most cost-beneficial to improve the I-cache and D-cache sizes.

As an example, Fig. 17 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.

7. Discussion

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 [24], reinforce this emerging new trend. WebCore can be integrated with other heterogeneous proposals that improve Web browsing efficiency (such as via scheduling [78]).

Parallelism None of the mainstream Web browsers, such as Chrome and Firefox, explicitly parallelize browser 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%.

Longevity As we see it, the longevity of the WebCore lies in the following aspects. First, the Web browser is, and will continue to be, the substrate of many Web applications due to its "write-once, run-anywhere" feature. Exemplifying this design philosophy is Google's recent announcement of the Portable Native Client (PNaCl). It supports the porting of native C/C++ applications to the Chrome browser [25]. SkyFire Technology shows that Web applications based on browser technologies still far outweigh native apps, excluding games [26]. 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 [27] make it possible to port entire real-time

physics engines such as the Unreal Engine into a browser [28].

Second, the kernels, their data structures, and the communication patterns that we study in this paper are not specific to a particular browser implementation. They 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.

8. Related Work

Web browser optimizations Zhu and Reddi propose scheduling to leverage the big/little heterogeneous system for optimizing the energy efficiency of mobile Web browsing [78]. WebCore customization and specialization enriches the heterogeneity at the core and system level, thus creating more opportunity for the compiler and operating system scheduling.

Prior software work focuses on parallelizing browser kernels/tasks for improving performance [29, 32, 41, 59, 62–64]. Our optimizations target both performance and energy, and can therefore readily improve the per-thread/task energy efficiency.

Seemingly similar to our work, SiChrome [34] performs aggressive specializations that map much of the Chrome browser into silicon. The key difference is that we retain general-purpose programmability while still being energy efficient.

Other works take a system-level perspective on improving Web browsing performance, such as asynchronous rendering, resource prefetching, and refactoring JavaScript and CSS files [58, 73–75, 77]. Our work is complementary to them since they can all benefit from kernel-level efficiency improvements.

Web browser workload characterization BBench [47] 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.

MobileBench [65] characterizes the performance impact of various microarchitecture features on mobile workloads. Our paper quantifies the performance-energy trade-off. MobileBench results show that more aggressive customizations (e.g., prefetcher) of general-purpose cores are worth exploring.

Other works analyze webpage-inherent characteristics to show the variances across different webpages [36, 70, 78]. They imply the potential of adaptive and dynamic specialization techniques, which are beyond the scope of our current work.

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 ($\sim$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 [45, 60, 61]. They are separate and complementary to our work involving the browser engine.

Specialization alternatives L0 caches and scratchpad memories [33, 52] 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 [30] 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 [44, 53]. The streaming access pattern of the DOM tree shown in Fig. 13 indicates that a dynamic cache insertion policy such as DIP [68] or an intelligent linked data structure prefetcher [43] 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.

9. Conclusion

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.

References

[1] Mobile Users Prefer Browsers over Apps. http://goo.gl/oZXZ7g
[2] Anonymized for blind review.
[3] The Benefits of Multiple CPU Cores in Mobile Devices. http://goo.gl/83j6zo
[4] The Benefits of Quad Core CPUs in Mobile Devices. http://goo.gl/A7e6Jc
[5] Fact Sheet: Gomez Mobile Monitoring. http://www.ndm.net/apm/pdf/Mobile_Monitoring_FS.pdf
[6] Battery Statistics. http://goo.gl/UZ9V4q
[7] Samsung Exynos 5 Octa. http://goo.gl/HwKJ8g
[8] Chromium. http://www.chromium.org/Home
[9] WebKit. http://www.webkit.org
[10] Alexa - The Web Information Company. www.alexa.com
[11] You can't get away from a bad mobile experience. http://goo.gl/5aWFwb
[12] The Relationship Between Faster Mobile Sites and Business KPIs. http://goo.gl/efX2Zy

[13] 2012 Web predictions. http://goo.gl/Bv6IE
[14] ARM Cortex A7. http://goo.gl/0SeJL
[15] Intel Atom Processor Z2460. http://goo.gl/TuEfD
[16] ARM Cortex A15. http://goo.gl/kJ4h
[17] Silvermont. http://goo.gl/TDRMTT
[18] Snapdragon SoC Wiki. http://goo.gl/KXFFh
[19] Exynos SoC Wiki. http://goo.gl/GJcuk
[20] ARM DS-5. http://ds.arm.com/ds-5/optimize/
[21] CSS Cascading Order. http://goo.gl/GQxujo
[22] CACTI 5.3. http://www.hpl.hp.com/research/cacti
[23] Teardown: Samsung Galaxy S4. http://goo.gl/f07jS5
[24] Hardware Support for WebRTC in Tegra4. http://goo.gl/AiTDxd
[25] Portable Native Client. http://goo.gl/9IFLIH
[26] Why Flurry Got It Wrong On Mobile Apps Vs. Web Browsers. http://goo.gl/lg52N
[27] Construct2. https://www.scirra.com/construct2
[28] Mozilla And Epic Games Bring Unreal Engine 3 To The Web. http://goo.gl/3cSsog
[29] Servo. https://github.com/mozilla/servo
[30] Krait Cache and Memory Hierarchy. http://goo.gl/QbZ8jv
[31] O. Azizi, A. Mahesri, B. Lee, S. J. Patel, and M. Horowitz, "Energy performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proc. of ISCA*, 2010.
[32] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum, "Towards parallelizing the layout engine of firefox," in *Proc. of USENIX HotPar*, 2010.
[33] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. of CODES+ISSS*, 2002.
[34] V. Bhatt, N. Goulding-Hotta, Q. Zheng, J. Sampson, S. Swanson, and M. B. Taylor, "Sichrome: Mobile web browsing in hardware to save energy," *DaSi: First Dark Silicon Workshop*, 2012.
[35] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *Proc. of HPCA*, 2013.
[36] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Understanding website complexity: measurements, metrics, and implications," in *Proc. of IMC*, 2011.
[37] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," in *Proc. of ISCA*, 1991.
[38] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. of USENIX ATC*, 2010.
[39] ——, "The systems hacker's guide to the galaxy," in *Proc. of APSYS*, 2013.
[40] C. Cascaval, S. Chatterjee, H. Franke, K. Gildea, and P. Pattnaik, "A taxonomy of accelerator architectures and their programming models," in *IBM Journal of Research and Development*, 2010.
[41] C. Cascaval, S. Fowler, P. M. Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar, "Zoomm: A parallel web browser engine for multicore mobile devices," in *Proc. of PPoPP*, 2013.
[42] G. Dunteman, *Principal Component Analysis*. Sage Publications, 1989.
[43] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *Proc. of HPCA*, 2009.
[44] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao, "Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms," in *Proc. of DAC*, 2011.
[45] L. Guckert, M. O'Connor, S. K. Ravindranath, Z. Zhao, and V. J. Reddi, "A case for persistent caching of compiled javascript code in mobile web browsers," in *Workshop on AMAS-BT*, 2013.
[46] Q. Guo, T. Chen, Y. Chen, Z. Zhou, W. Hu, and Z. Xu, "Effective and efficient microprocessor design space exploration using unlabeled design configurations," in *Proc. of IJCAI*, 2011.
[47] A. Gutierrez, R. Dreslinski, A. Saidi, C. Emmons, N. Paver, T. Wenisch, and T. Mudge, "Full-system analysis and characterization of interactive smartphone applications," in *Proc. of IISWC*, 2011.
[48] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proc. of ISCA*, 2000.
[49] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proc. of ISCA*, 2010.
[50] F. E. Harrell, *Regression Modeling Strategies*. Springer, 2001.
[51] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," in *Proc. of MobiSys*, 2012.
[52] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proc. of MICRO*, 1997.
[53] T. Kluter, P. Brisk, E. Charbon, and P. Ienne, "Way stealing: A unified data cache and architecturally visible storage for instruction set extensions," in *IEEE Transactions on VLSI*, 2013.
[54] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. of ASPLOS*, 2006.
[55] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of MICRO*, 2009.
[56] Y. Li, M. Hempstead, P. Mauro, D. Brooks, Z. Hu, and K. Skadron, "Power and thermal effects of sram vs. latch mux design styles and clocking gating choices," in *Proc. of ISLPED*, 2005.
[57] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A low-power architecture for software radio," in *Proc. of ISCA*, 2006.
[58] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "Pocketweb: instant web browsing for mobile devices," in *Proc. of ASPLOS*, 2012.
[59] H. Mai, S. Tang, S. T. King, C. Cascaval, and M. Pablo, "A case for parallelizing web pages," in *Proc. of USENIX HotPar*, 2012.
[60] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism," in *Proc. of HPCA*, 2011.
[61] M. Mehrara and S. Mahlke, "Dynamically Accelerating Client-side Web Applications through Decoupled Execution," in *Proc. of CGO*, 2011.
[62] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proc. of WWW*, 2010.
[63] ——, "Ftl: Synthesizing a parallel layout engine," in *Eucopean Conference on Object-Oriented Program*, 2012.
[64] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik, "Parallel schedule synthesis for attribute grammars," in *Proc. of PPoPP*, 2013.
[65] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite – mobilebench," in *Proc. of IISWC*, 2013.
[66] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *Proc. of DAC*, 2011.
[67] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. of ISCA*, 2013.
[68] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of ISCA*, 2007.
[69] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Characterizing real-world behavior of javascript programs," in *Technical Report MSR-TR-2009-173, Microsoft Research*, 2009.
[70] A. Sampson, C. Cascaval, L. Ceze, P. Montesinos, and D. S. Gracia, "Automatic discovery of performance and energy pitfalls in html and css," in *Proc. of IISWC*, 2012.
[71] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, "Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound," in *Proc. of HPCA*, 2013.
[72] F. Schlachter, "No Moore's Law for Batteries," in *Proc. of National Academy of Science of the United States of America*, 2013.
[73] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who killed my battery?: analyzing mobile browser energy consumption," in *Proc. of WWW*, 2012.
[74] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie, "Why are Web Browsers Slow on Smartphones?" in *Proc. of HotMobile*, 2011.
[75] ——, "How Far Can Client-Only Solutions Go for Mobile Browser Speed?" in *Proc. of WWW*, 2012.
[76] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Anysp: Anytime anywhere anyway signal processing," in *Proc. of ISCA*, 2009.
[77] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proc. of WWW*, 2010.
[78] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proc. of HPCA*, 2013.