Energy-Efficient Mobile Web Computing

Ph.D. Qualifying Examination Proposal

Yuhao Zhu

Electrical and Computer Engineering Department
The University of Texas at Austin
yzhu@utexas.edu
http://yuhaozhu.com/

Abstract

Next-generation Web services will be primarily accessed through mobile devices. However, mobile devices are low-performance and stringently energy-constrained. In this thesis, I propose the design of a high-performance and energy-efficient mobile computing substrate. It is a hardware/software co-designed system that delivers satisfactory user quality-of-service (QoS) experience on a mobile energy budget. The key insight that the traditional interfaces between different Web stacks need to be enhanced with new abstractions that express user QoS experience and that expose architectural-level complexities. On the basis of the enhanced interfaces, I propose synergistic cross-layer optimizations across the processor architecture, Web runtime, programming language, and application layers to maximize the whole system efficiency. The proposal is likely to have a long-term impact because the target application domain, the Web, is becoming a universal mobile development platform, and our solutions target the fundamental computation layers of the Web domain.

1 Introduction

Web technologies have shaped how we think, communicate, and innovate. Over the past decade, the Web's role shifted from solely information retrieval (Web 1.0) to providing interactive user experiences (Web 2.0). Now the Web is once again on the cusp of a new evolution that features automatic recognition, mining, and synthesis of user-originated "big data." The driving force behind this evolution is today's most pervasive personal computing platform—mobile devices. It is estimated that 3 billion Web-connected mobile devices currently exist, and will reach nearly 50 billion by 2020 [48]. Next-generation Web services will be primarily accessed through mobile devices as opposed to desktops and laptops as in previous generations [61].

While there is a significant opportunity for growth, technology challenges in current mobile systems could impede the impact of future Web services. Mobile devices are low-performance, stringently energy-constrained, and rely solely on wireless connections, the combined effect of which often leads to poor quality-of-service (QoS) experience. Mobile users react to poor QoS experience by abandoning Web services. For example, 25% of mobile users abandon webpages that take over 4 seconds to load [60]. Google estimated that "a 400 ms delay leads to a 0.44% drop in search volume." [57] Web service abandonment due to poor mobile QoS experience has become a major roadblock to the growth and adoption of next-generation Web computing.

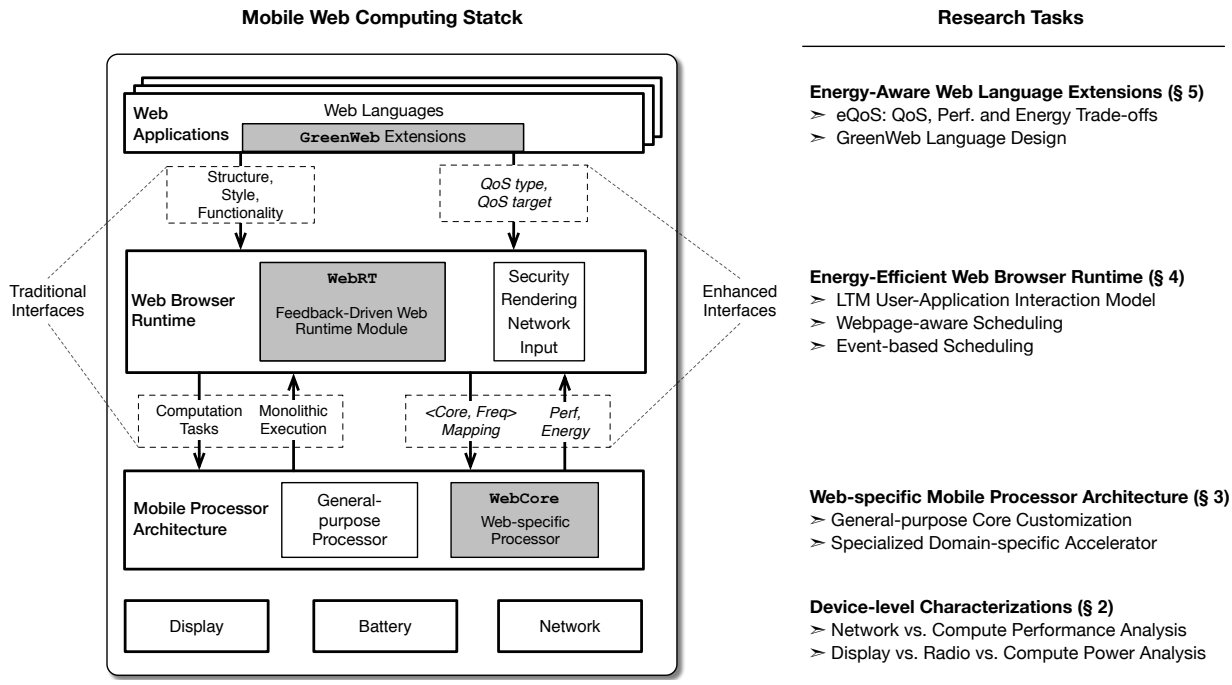

Fig. 1: Overview of the cross-layer research contributions.

My Ph.D. research objective is to design a next-generation mobile computing substrate—a hardware/software collaborative system that is energy-efficient and delivers satisfactory user QoS experience. Given the board scope of mobile Web that involves both computation and network, I first quantify the impact of computation and network on mobile Web [92]. I find that generational advancements in cellular network technology have reached a point where further improving the network latency only leads to marginal performance improvement with prohibitively high energy consumption. In contrast, the compute starts having a significant impact on mobile Web performance and energy consumption. Therefore, I focus my research on the *computation* side.

I take a holistic view of the mobile Web computation stack, spanning applications, Web browser runtime, and processor architecture. Prior art has been mostly focused on optimizations within individual layers while maintaining traditional interfaces to other layers. My preliminary work [93, 97, 92, 94, 91, 95, 53, 96] has demonstrated that improving energy efficiency while continuing to scale performance of the mobile Web requires us *to enhance the traditional interfaces with new abstractions and to leverage the new interfaces for cross-layer optimizations*.

At the application/Web browser boundary, current Web applications merely specify visual appearance and functionalities to the browser through Web languages such as HTML, CSS, and JavaScript. User QoS requirements (e.g., latency tolerance) are unexpressed. However, different users QoS requirements lead to different optimal runtime decisions for trading off QoS with energy consumption. Exposing user QoS expectations at the application level would allow the Web runtime to budget wisely the energy usage while delivering satisfactory user QoS experience.

At the Web browser/architecture boundary, the traditional interface provides to the Web browser runtime a simple sequential execution model of the hardware. However, today's mobile processors architectures are becoming extremely complex, combining general-purpose cores that have different performance and energy characteristics [62] with special-purpose domain-specific ac-

celerators. While the hardware upheaval promises performance and energy improvements for the mobile Web, its practical impact depends on how effective the Web browser can leverage it. I see needs on both improving the processor architecture for the Web domain and designing an intelligent Web browser runtime that can effectively manage the hardware complexities.

In the spirit of enhancing the traditional Web computing stack interfaces and leveraging the new interfaces to optimize each layer, I propose the following research items. Fig. 1 gives an overview of the proposed cross-layer research. Enhancements to the existing Web stack are shaded.

- **Web Language Extensions:** I propose `GreenWeb`, a set of language extensions that let Web developers express user QoS expectations as program annotations. `GreenWeb` is based on two new programming abstractions, QoS type and QoS target, that capture two fundamental aspects of user QoS experience. `GreenWeb` does not pose any constraints on specific runtime implementations but instead supports general energy optimization techniques.
- **Smart Web Browser Runtime:** I propose `WebRT`, a mobile Web browser runtime that optimizes for energy-efficiency while delivering the specified user QoS requirements. Although `WebRT` is a generic runtime design, as a prototype implementation, I propose to leverage the big-little architecture as the hardware substrate, which enables `WebRT` to tune dynamically the hardware configurations according to user QoS requirements for energy optimizations. `WebRT` also continuously monitors hardware execution to enable adaptive optimizations.
- **Web-Specific Processor Architecture:** I propose `WebCore`, a forward-looking mobile CPU architecture customized and specialized for the Web stack. The `WebCore` improves performance and energy-efficiency simultaneously by integrating domain-specific hardware that exploits critical computation kernels. `WebCore` also maintains general-purpose programmability, which is vital to ensure its applicability to the complex Web software stack.

The *long-term impact* of my proposal lies in two fundamental aspects. First, performance and energy will continue to be the primary constraints on mobile devices as they evolve in the future (e.g., wearables and Internet-of-Things (IoT) devices). Therefore, the problem I study is likely a long-term challenge for mobile computing. Second, the Web has been and will continue to be a universal application platform because it enables application portability to tackle the notorious device fragmentation issue [74]. The proposed techniques target fundamental computations layers of Web technologies. Thus, they are likely to have long-term applicability.

The rest of the proposal is organized as follows. Sec. 2 quantitatively demonstrates the need for high-performance and energy-efficient computation in the mobile Web. It directly motivates the research theme of my proposal. Sec. 3, Sec. 4, and Sec. 5 describe the proposed `WebCore`, `WebRT`, and `GreenWeb` at the architecture, runtime, and application layer, respectively. Sec. 6 provides a retrospective and prospective view of the proposed work. The retrospective part summarizes the principles distilled from the work so far about building an energy-efficient mobile Web computing system; the prospective part suggests next steps for generalizing the principles and outlines specific research items planned in preparing the final dissertation.

2 The Need for High-Performance and Energy-Efficient Computation in Mobile Web

This section quantitatively demonstrates the importance of *computation*, among other components such as network and display, to mobile Web's performance and energy consumption. The observations discussed in this section directly motivate my research to focus on the computation layer of the mobile Web and to improve its performance and energy-efficiency.

The computation layer involves many mobile SoC components, such as CPU, GPU, and domain-specific accelerators. My proposal specifically focuses on the CPU for the following two reasons. First, CPU is the most heavily exercised computation component for Web applications because the Web runtime primarily targets CPUs. GPUs' usage, although providing critical performance benefits, is still limited to specific tasks such as rasterization and compositing [12]. The key computations such as layout and JavaScript execution are still solely performed on general-purpose CPUs. Second, CPU serves as an incubator for future accelerators—we must first understand computation kernels' characteristics on CPUs before they can be accelerated.

In the rest of this section, I first show that mobile Web performance increasingly depends on the computational capability of mobile CPUs, indicating the need for a high-performance computation (Sec. 2.1). I then show that mobile devices' power consumption is increasingly dominated by CPUs, calling for an energy-efficient computation (Sec. 2.2). Note that all the results presented in Sec. 2.2 are adapted from a recent research project [53] that I collaborated on and contributed to.

2.1 The Importance of High-Performance Computation in Mobile Web

Computation and network largely dictate the performance of mobile Web. Conventional wisdom suggests that mobile Web performance is primarily limited by the network latency. In this section, I quantify the impact of CPU and network performance by experimentally comparing how the webpage load time varies with different CPU and network performance on today's high-end smartphone Galaxy S5. I show that as cellular network technologies evolve over generations, mobile Web performance becomes sensitive to CPU performance.

Network Impact To study the impact of network latency of various cellular network generations, we host all the webpages on a Web server, into which we manually inject delay. We then use Wi-Fi on the smartphone to access the webpages. Since Wi-Fi has significantly lower latency than the current 4G/LTE network, the delay injection lets us mimic a wide range of network latencies.

Holding the CPU performance at its peak, Fig. 2 shows the webpage load time with respect to different network latencies. We superimpose the figure with different mobile network technologies' typical latencies derived from both technical specifications as well as real measurements in the field [50, 4]. We observe that reducing the network latency from an adverse 3G connection at 2,000 ms to an LTE connection at 100 ms results in a 9.5X speedup in webpage load time from 38 seconds to 4 seconds. As the network latency further improves within the range of LTE network latency (50∼100 ms), the network latency has only a marginal impact on the overall webpage load time. This is because at this point the fast network accesses are hidden behind CPU computations in the asynchronous execution model; the application is largely CPU-bound. Further reducing the network latency from LTE to Wi-Fi has almost no effect.

Computation Impact As the network latency becomes low (e.g., under the LTE technology), the CPU performance starts playing a significant role in the mobile Web performance. To study how the CPU performance affects the webpage load time, we mimic a wide range of CPU performance capabilities by leveraging S5's 14 frequency settings. Note that we use frequency only as a *proxy*

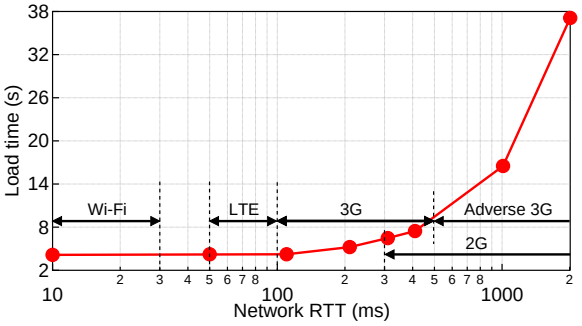

Fig. 2: Webpage load time with respect to changing network latency. Each marker corresponds to an RTT value. We also superimpose the round-trip time (RTT) range for different cellular technologies derived from both technical specifications as well as real measurements in the field [50, 4].

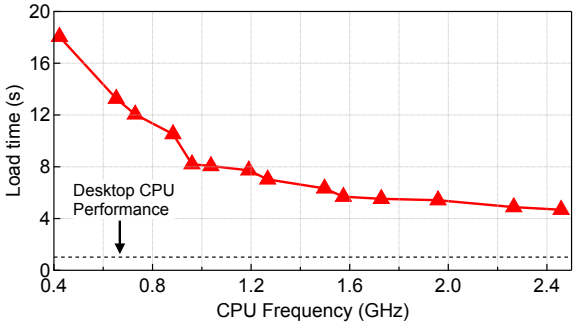

Fig. 3: Webpage load time with respect to CPU frequency on a Galaxy S5 smartphone. The markers represent CPU frequencies, which range from 0.4 GHz (left) to 2.5 GHz (right). We also overlay the load time on a desktop CPU (the dotted line) for comparison purpose.

for CPU performance, it is *not* our intention to study the impact of a particular CPU's frequency itself. Fig. 3 shows how webpage load time changes with CPU performance under a 100 ms RTT (LTE-like cellular network connectivity). As the CPU frequency decreases from the highest to the lowest by about 6X (2.5 GHz to 0.4 GHz), the webpage load time slows down by as much as 4.5X from 4 seconds to about 18 seconds, indicating strong sensitivity to CPU performance.

Note that increasing clock frequency between 1.6 GHz and 2.4 GHz yields small performance benefits. One may then naively conclude that mobile CPU performance improvements provide marginal improvements in Web performance. However, the "marginal improvement" is merely an artifact of using frequency as a performance proxy. At high frequencies, the processor's pipeline is already saturated, and the memory and interconnection become the microarchitectural-level bottlenecks [28]. To overcome this artificial constraint and assess the impact of future mobile CPU improvements, we perform the same experiment on a desktop CPU (Intel Core i5 at 1.2 GHz). The result is shown as the dotted line in Fig. 3. The average webpage load time on the desktop CPU is about 1 second, effectively a 4X speedup over the peak performance of S5. This experiment shows that mobile CPU performance today is still far from reaching a diminishing return point, and it can continue to have a significant impact on mobile Web performance.

The takeaway from the results is that continuous improvement to network latency will eventually, if not already, take us to a point where further Web performance improvement will be unattainable without improving CPU performance. This is a timely conclusion, especially when low latency cellular network, such as LTE, is already prevalent today. It is estimated that LTE's subscription will reach 1.37 billion (one-fifth of the world population) by the end of 2015 [15].

2.2 The Need for Energy-Efficient Computation in Mobile Web

Despite the need for high-performance computation, mobile devices are severely limited by a battery-imposed energy budget, which in turn limits the achievable performance. In this section, I first use smartphones to quantitatively demonstrate that the energy budget of mobile devices is likely to stay stringently constrained in the near future. I then show that the CPU is becoming the worst power and energy consumer of a mobile device as compared to other components such as display and radio. There is clearly a need for energy-efficient computation in the mobile Web. Data presented here is adapted from the results of a related project [53] that I collaborated on.

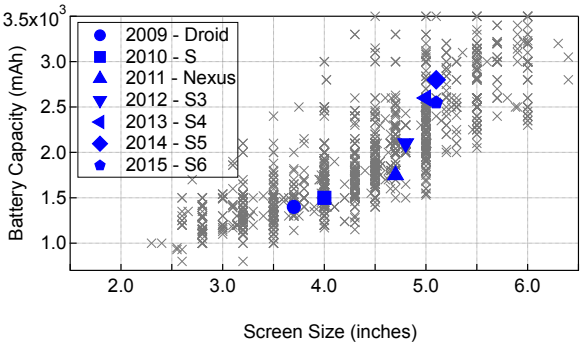

Fig. 4: There is an almost linear relationship between battery and screen size over the time.

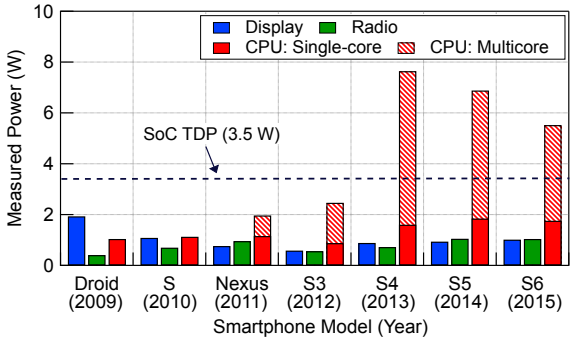

Fig. 5: CPU power consumption has increased significantly compared to other key components.

Energy Constraint Battery technology has not experienced Moore's law-like improvements because of fundamental physics limitations [78]. As a result, the density of lithium-ion batteries has improved by only about 10% per year [34]. Therefore, the battery capacity of today's mobile devices is determined by the battery's volume, which is largely dictated by the device's screen size [5]. Using the smartphone as an example of a start-of-the-art mobile device, Fig. 4 compares the screen sizes and battery capacities of over 600 smartphones from 2006 to 2014. There is a near-linear correlation between the battery capacity and screen size. As smartphone form factors reach maturity [19], the total device energy budget will likely stay severely constrained.

Mobile CPU's Rise to Power Different components contribute to the overall power consumption of a mobile device. Fig. 5 compares the measured power consumption of three major mobile device components: CPU, display, and radio. We select seven top smartphones for each year from 2009 to 2015. They are Motorola's Droid from 2009, and Samsung's Galaxy S, Nexus, S3, S4, S5, and S6 from 2010 through 2015, respectively. Chronologically, the seven phones represent how cutting-edge smartphone technologies have progressed over time. The results are collected while running a standard Web benchmark, Sunspider [21], for the CPU(s), and dedicated benchmarks for the other components [39, 17]. We used the Monsoon power monitor to measure the seven smartphones' power consumptions at the battery level.

We make two important observations from Fig. 5. First of all, CPU is now a major power consumer of a mobile device. The year 2011 marks an inflection point where a single CPU core began overtaking the display as the most power consuming component. On any of the last three mobile CPU generations, the multicore CPU can exceed the entire mobile device's thermal design power (TDP) even without including the radio and display's power consumptions.

Second, while other components are becoming more power-efficient over time due to technological advancements [42], the CPU's power has risen sharply. The excessive mobile CPU power consumption is a direct result of current mobile CPU design strategy, i.e., merely adopting desktop-like design techniques such as aggressive single-core microarchitecture enhancement and multicore scaling to improve performance [53]. As the Dennard Scaling [45] comes to its end, the performance benefits do not sufficiently make up for the additional power consumption [53], eventually hurting mobile CPU's energy-efficiency.

Given that users expect each mobile device generation to incorporate new peripherals such as sensors that also require energy from the same budget, it is clear that the CPU, as a major energy consumer, needs to become more energy-efficient while sustaining performance improvement.

3 WebCore: Web-Specific Mobile Processor Architecture

Domain-specific specialized architecture 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 [54, 65, 84]. 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 [58]. Recent work has demonstrated the importance and feasibility of balancing general-purpose programmability and specialization in various data computation domains (e.g., H.264 encoding [54], convolution [75]).

Following the same architecture design philosophy of balancing general-purpose programmability and domain-specific specialization, I propose the `WebCore`, a general-purpose CPU customized and specialized for Web technologies. `WebCore`'s design starts from existing mobile CPUs, and thus retains the general-purpose programmability. It achieves performance and energy improvement by combining customization and specialization techniques. In the rest of this section, I describe the customization and specialization process in Sec. 3.1 and Sec. 3.2, respectively. Sec. 3.3 discusses prior work on hardware support for mobile Web.

3.1 Customizing General-Purpose Cores

`WebCore` design is based on general-purpose CPUs to best retain the general-purpose programmability. However, existing general-purpose processors may not be an ideal baseline for `WebCore`, because they are not uniquely tuned for Web applications. `WebCore` customizes current designs by exploring a vast design space to properly size key microarchitecture parameters.

I derive two major conclusions through the customization process. First, out-of-order designs provide more flexibility for energy versus performance trade-offs than in-order designs. Second, a customized out-of-order design configuration still contains two sources of inefficiency—instruction delivery and data feeding—that need to be further mitigated.

Design Space Exploration (DSE) We mine through the top 10,000 websites as ranked by Alexa [29] and leverage the principal component analysis (PCA) to select the 12 most representative websites. All except one happen to rank among Alexa's top 25 websites. We consider both mobile and desktop versions of the 12 websites because tablets typically load the desktop version of webpages. In total, we study 24 distinct webpages.

We consider various tunable microarchitectural parameters such as issue width, instruction/data cache sizes. We vary the values of functionally related parameters (e.g., issue width and the number of functional units) together to avoid reaching an entirely unbalanced design [37]. We also do not consider single-issue out-of-order processors, which are known to be energy inefficient [31]. In total, we consider over 3 billion design points.

It is not feasible to simulate all of the design points simply due to time constraints. Therefore, we leverage regression modeling techniques [55] 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 [63, 51]. We construct performance and power models for in-order and out-of-order designs separately. Overall, the out-of-order models' error rates

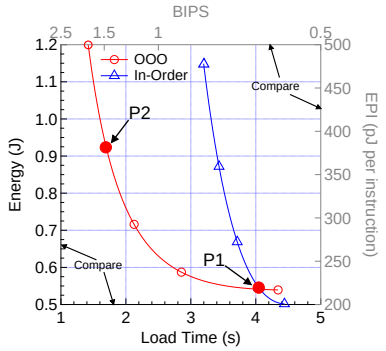

Fig. 6: In-order versus out-of-order Pareto optimal frontiers.

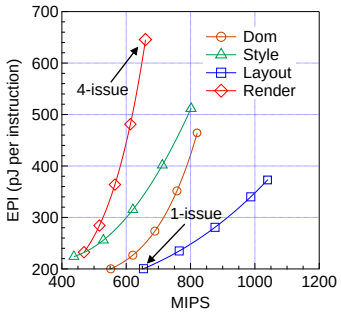

Fig. 7: In-order Pareto optimal frontier for each kernel.

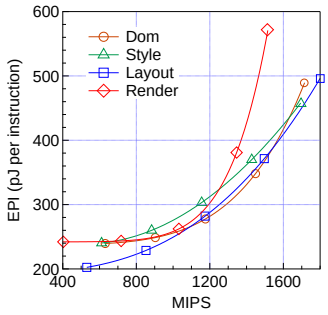

Fig. 8: Out-of-order Pareto optimal frontier for each kernel.

are below 6.0%. The in-order performance and power models' errors are within 5% and 2%, respectively. The in-order models are more accurate because of their simpler design.

Core Choice 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. Fig. 6 shows the Pareto optimal frontiers of the in-order and out-of-order design space. We observe that the in-order design space has a narrow performance range of 1 second, whereas the out-of-order space covers a 4 second performance range. The performance range contrast indicates that out-of-order designs can more flexibly balance performance with energy and, therefore, are better design baselines for mobile Web applications.

To understand the difference behind the in-order versus out-of-order designs, we study the kernel behaviors in Web applications. There are four important computation kernels in executing a Web application: i.e., *Dom*, *Style*, *Layout*, and *Render*. They contribute to about 75% of the webpage load time and energy consumption. Fig. 7 and Fig. 8 show the Pareto optimal frontiers of the in-order and out-of-order design space for each kernel. We find that the kernel variance in the in-order designs is more pronounced than in the out-of-order designs. As we push toward more performance in the in-order design space, some kernels stop scaling gracefully on the energy-versus-delay curve, and eventually become a performance bottleneck. Overall, in-order designs have low marginal performance value with high marginal energy cost [31]. In contrast, out-of-order cores can cover the variances across the different kernels through complex execution logic and, therefore, provide wider performance and energy trade-off range.

Sources of Inefficiency DSE also helps customization at the microarchitecture level. We examine microarchitectural parameters of two out-of-order Pareto optimal designs: P1 and P2 in Fig. 6. They represent designs optimized for different performance and energy targets. P1 is optimized for minimal energy consumption in the out-of-order space. P2 is a high-performance design with a performance of 1500 MIPS (million instructions per second). Table 1 summarizes the microarchitecture configurations of the two designs. For comparison purposes, it also lists the same parameters for ARM Cortex-A15, which represents today's high-end mobile CPU.

By comparing P1 and P2 with Cortex-A15, we find two major sources of inefficiencies in general-purpose processors: instruction delivery and data feeding. First, current mobile processors have a small L1 instruction cache that is typically 32 KB in size. However, the two Pareto optimal

Table 1: Microarchitecture configurations for P1 and P2 in Fig. 6. They represent different energy-delay trade-offs. For comparison purpose, we also show the parameters for ARM Cortex-A15, whose information is gather from measurements using the 7-Zip LZMA Benchmark [27] and ARM's public presentation [30].

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

designs require a 64 KB to 128 KB instruction cache to alleviate the pressure on instruction delivery in mobile Web applications. The pathological front-end behavior mainly stems from the large instruction footprint and the prevalence of the irregular control flow path [52].

Second, the high-performance design P2 also necessitate a 64 KB data cache, doubling the typical L1 data cache size in current mobile CPUs. The need for a large data cache mainly stems from the large working set size on principal data structures (e.g., the DOM tree) during webpage processing. For example, profiling results show that the average data reuse distance for DOM tree accesses is 4 KB (excluding other memory operations interleaved with DOM accesses). The large data cache leads to excessive energy consumption and needs to be optimized.

3.2 Specializing the Customized Cores

Unusual design parameters in a customized processor tuned for the mobile Web workload indicate that instruction delivery and data feeding are critical to guarantee high performance while still being energy efficient. I propose a specialized functional unit that mitigates the two bottlenecks. The new hardware structure is accessed via a set of high-level language APIs implemented as a runtime library to maintain the general-purpose programmability.

Style Resolution Unit The *Style* kernel is an important computation kernel in a Web browser [94]. It takes about one-third of the total execution time and energy consumption of webpage loading. Therefore, optimizing the *Style* kernel would provide the most overall benefits. In order to mitigate the instruction delivery and data communication overhead of the *Style* kernel, I propose a hardware functional unit called the *Style Resolution Unit* (SRU). 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. To exploit the inherent fine-grained parallelism, the SRU employs a multi-lane parallel architecture, which

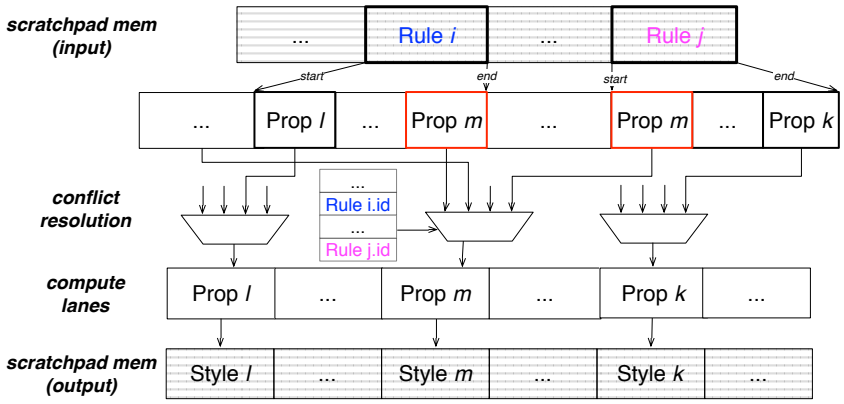

Fig. 9: SRU coupled with scratchpad memories.

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.

The *Style* kernel takes a set of input CSS rules, and applies the rules in certain cascading order [83] to calculate a final value of each CSS property (e.g., the exact RGB values for the `color` property, or the number of pixels for the `width` property). These values are stored back to a data structure called render tree for further processing by other kernels.

The key observation we make from the *Style* kernel algorithm is that there are two types of inherent parallelism: "rule-level parallelism" (RLP) and "property-level parallelism" (PLP). Improving the performance and energy consumption of the *Style* kernel requires us to exploit both forms of parallelism. RLP comes from the following observation. In the sequential algorithm, each rule must be iterated from the lowest priority to the highest following the cascading order [83], so that values from higher-priority rules can override values from lower-priority rules. However, we could speculatively apply rules with different priorities in parallel, and simply select the correct value according to the rule priorities in the end. PLP follows RLP. Each rule contains value information of multiple CSS properties. Calculating values for all the properties contained within a rule is independent of each other, and can be executed in parallel.

We propose Style Resolution Unit (SRU) as a parallel hardware unit to exploit both RLP and PLP. The SRU aggregates enough computations to reduce control-flow divergences and increase arithmetic intensity. It is accompanied by scratchpad memory that reduces the overhead of data communication for both input and output. Fig. 9 shows the structure of the SRU with scratchpad memory. SRU is a SIMD-like structure with multiple compute lanes, with each lane calculating the final value of 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 Rule i because it has a higher priority. The compute lanes then write the final values back to the output scratchpad memory.

Implementation Details 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. We profile the webpages to determine the appropriate amount of resource allocation. Profiling indicates that

90% of the time, the RLP is below or equal to 4. 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. Thus, we implement a 32-wide SRU where each lane handles one hot CSS property.

Due to these considerations, the scratchpad memories are each 1 KB in size. Under Synposys 28 nm toolchain, the total area overhead of SRU is about 0.59 mm^2, negligible compared to typical mobile SoC size (e.g., Samsung's Exynos 5410 SoC has a total die area size of 122 mm^2 [88]). The SRU logic introduces 70 mW total power, which is insignificant compared to power consumption in a typical mobile CPU (e.g., about 1 W on a single core Cortex-A15). The scratchpad memory can be accessed in one cycle, which is the same as the fastest L1 cache latency in our design space. The SRU logic latency is about 16 cycles under 1.6 GHz.

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

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. 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. For properties that can be 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.

3.3 Related Work

Similar to `WebCore`, SiChrome [35] performs aggressive specializations that map much of the Chrome browser into silicon. The key difference is that `WebCore` starts from a (well-optimized) general-purpose baseline and thus retains general-purpose programmability while still being energy-efficient. In addition, SiChrome evaluates energy-efficiency using the EDP metric while our Pareto optimal analysis provides a more generic optimization view than EDP.

EFetch [40] and ESP [41] also propose specialized hardware structures on top of general-purpose cores to improve the performance and energy-efficiency of Web applications. They view a Web application execution as a sequence of events. As a result, the proposed specialized hardware primarily targets the inefficiencies associated with the event-driven execution model. `WebCore` views a Web application execution as a mix of different kernels. As such, the proposed specialization technique targets individual kernels. Both views are complementary in that per-event execution can benefit from kernel-level improvement that `WebCore` provides and vice versa.

4 WebRT: Smart Web Browser Runtime Optimizing for Energy-Efficiency

Today's mobile processors are becoming extremely heterogeneous. They often combine general-purpose cores that have different performance and energy characteristics [62] (e.g., asymmetric chip-multiprocessor architecture) with special-purpose domain-specific cores (e.g., `WebCore`). While the hardware upheaval promises performance and energy improvements for the mobile Web, current Web runtime systems are not designed to fully exploit the capability of the underlying hardware. The main bottleneck is that current runtime-architecture interface merely exposes the hardware as a monolithic sequential execution model to the runtime system while hiding many architecture-level details. Without having a full visibility of the hardware details, current Web runtimes often lead to energy-inefficient decisions or violate user QoS requirement.

To bridge the widening gap between the architecture complexity and the architecture-agnostic runtime system, I propose to enhance the existing runtime-architecture interface by exposing architecture details to the Web runtime. I specifically focus on the ACMP architecture [81, 62] as the hardware substrate. ACMP is long known to provide a large performance-energy trade-off space, and is already widely used in today's mobile systems [6, 13]. Leveraging the enhanced interface, I propose `WebRT`, a Web runtime that minimizes energy while guaranteeing satisfactory user QoS experience by scheduling Web application executions using proper ACMP configurations.

In the rest of this section, I first provide an overview of `WebRT` (Sec. 4.1). I discuss that optimizing energy consumption while delivering desirable user experience requires us first to understand the nature of different user interactions and to devise different optimization schemes accordingly. Specifically, I introduce a user-application interaction model called LTM. LTM captures three fundamental user interactions in mobile Web applications–Loading, Tapping, and Moving–and provides a framework for reasoning about different energy optimization strategies. I then describe `WebRT`'s two critical components. The first component is the webpage-aware scheduler that targets the loading (L) process of a Web application (Sec. 4.2), and the second component is the event-based scheduler that targets the touching (T) and moving (M) interactions (Sec. 4.3). Finally, I compare the contrast `WebRT` with prior work on software support for mobile Web (Sec. 4.4).

4.1 WebRT Overview

An ACMP consists of cores with different microarchitectures, such as out-of-order and in-order. Each core has a variety of frequency settings. Different core and frequency combinations provide a large trade-off space between performance and energy. The objective of our ACMP-based `WebRT` is to find an ideal ACMP configuration (i.e., a $\langle core, frequency \rangle$ tuple) that minimizes the energy consumption while guaranteeing an acceptable responsive time when a user interaction happens.

To systematically analyze user interactions in mobile Web applications, we introduce a simple conceptual model called LTM, which captures three primitive user interaction forms: loading application page (L), tapping the display (T), and moving a finger on the display (M). The three interactions cover a majority of human-computer interactions on mobile devices: every application requires a loading phase (L), and post-loading interactions on mobile devices are mostly performed in the form of finger tapping (T) or finger moving (M).

The prediction strategy for Loading needs to be different from that for Touching and Moving. The fundamental difference is that Loading occurs only once per usage session while Touching and Moving interactions occur repetitively throughout the entire Web application usage session. As a result, it is possible to make the prediction for the Touching and Moving interactions based on

Category	Model Predictors
Webpage primitive: HTML	Number of each tag Number of each attribute Number of DOM tree node
Webpage primitive: CSS	Number of rules Number of each selector pattern Number of each property
Content-dependent	Total image size Total webpage size

Table 2: Model Predictors

the history information within the same usage session. For Loading, however, every application loading is likely different from the previous one, and as such we can not make predictions based on previous loadings of (potentially different) applications. Instead, we have to make prediction based on the particular content of a given Web application. I now discuss the `WebRT` component that targets the Loading (Sec. 4.2) and Touching and Moving interactions (Sec. 4.3) separately.

4.2 Webpage-aware Scheduling

In this section, we show that it is possible to predict the webpage load time and energy consumption using merely webpage-inherent characteristics. This prediction scheme had two advantages. First, it does not rely on any previous webpage loading history information and is based completely on each webpage's inherent characteristics. Second, the prediction is performed at the webpage parsing time which happens at the very beginning of the loading process, and as such allows enough time for energy optimizations. Based on such predictions, we propose a webpage-aware scheduler as a `WebRT` component that predicts the ACMP configuration for webpage loading in order to minimize energy consumption while meeting a specified cut-off latency.

Model Derivation We find that regression models provide sufficient accuracy to predict the webpage load time and energy consumption. A regression model is a mathematical function between a set of predictors and a response. Within our context, the response is either the webpage's load time or energy consumption in loading the webpage. The predictors are a set of webpage characteristics. We also require a number of sampling observations to train the model. The linear regression is the basic regression technique and is the premise for advanced ones. Therefore, we first provide fundamentals of linear regression modeling. We then identify the predictors to the model and obtain a set of sampling observations in order to derive the linear model. After that, we describe how the different insights gathered in the previous section on webpage characteristics led us to refine the basic linear model.

The linear regression model models a webpage's load time and energy consumption (responses) as a linear combination of various webpage characteristics (predictors), formulated as: $y = \beta_0 + \sum_{i=1}^{p} x_i\beta_i$ 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 identify the best-fitting β that minimizes the residual sum of squares (RSS) [56].

We consider two types of predictors. The first type includes the *webpage-inherent* primitives such as the number of HTML tags. In addition, we must also consider the impact of *content-dependent characteristics* such as image size and the total size of a webpage. These characteristics are coarse-

grained metrics that are independent of webpage structures but which influence the load time and energy of rendering. We summarize these features in Table 2. In total, we consider 376 predictors. We require a number of sampling observations to construct the regression models. In total, we obtain 2,500 sampling observations, for which we measure both webpage load time and energy consumption simultaneously on the Cortex-A9 processor running at 1.2 GHz.

We apply a set of techniques to improve the accuracy of the basic linear model. The most significant one is based on the observation that the true relationship between the response and all predictors is strictly linear as assumed by simple linear models. One effective method to model nonlinearity is to fit data with *restricted spline* functions that are piecewise polynomial functions but which force linear fitting beyond the first and last knots [56].

Model Evaluation To validate the model, we obtain 2,500 observations in addition to the 2,500 observations used in deriving the model. Overall, the performance model predicts 73.0% of the webpages within 10% error and 94.0% webpages within 20% error. Similarly, the energy model predicts 70.0% of the webpages within 10% error and 91.8% webpages within 20% error. Overall, we find the model accuracy to be acceptable.

Scheduler During the parsing stage, the webpage-aware scheduler extracts webpage characteristics, and feeds them into the prediction models to estimate the webpage load time and energy consumption under different core and frequency configurations. On the basis of these predictions, the scheduler then identifies the configuration (if possible) that meets the cut-off latency with minimal energy consumption. If no such configuration is found, the webpage is scheduled to the big core with the highest frequency for the best possible performance.

Overheads The scheduler accounts for scheduling overheads, which consist of two components: the prediction overhead and the overhead of changing the architecture configuration (i.e, big/little core migration and/or frequency scaling). The prediction cost is negligible. We assume an 100 μs overhead for frequency scaling and a 20 μs overhead for core migration, as indicated in whitepapers released by ARM [24, 25]. Note that the overhead of switching architecture configuration also applies to the event-based scheduling discussed below.

4.3 Event-based Scheduling

I propose event-based scheduling (EBS) as the mechanism to optimize energy-efficiency for the Touching (T) and Moving (M) interactions. Each T or M interaction is internally translated to an application event. EBS is based on the observation that a T or M event may occur repetitively throughout a Web application usage session such that it is possible to predict the ideal architecture configuration of an event based on its history information of performance and energy consumption. I first provide an overview of EBS and discuss several key implementation details.

Overview The key idea of identifying an event's ideal execution configuration is to build a performance model and an energy model. They predict an event's latency and energy consumption under any core and frequency combination. With the two models, EBS sweeps all possible core and frequency combinations and selects the one that meets the QoS target with minimal energy.

The scheduler consists of a simple dispatch frontend and scheduling backend as illustrated in Fig. 10. The frontend *Dispatch* unit extracts relevant event information, and passes it to the backend. The backend consists of a *Detector*, a *Model Constructor* and a *QoS Monitor*. The detector automatically identifies each event's QoS requirement. In its simplest form, the detector assumes a default latency target, such as 100 ms, for each event. If an event is annotated with programmer-

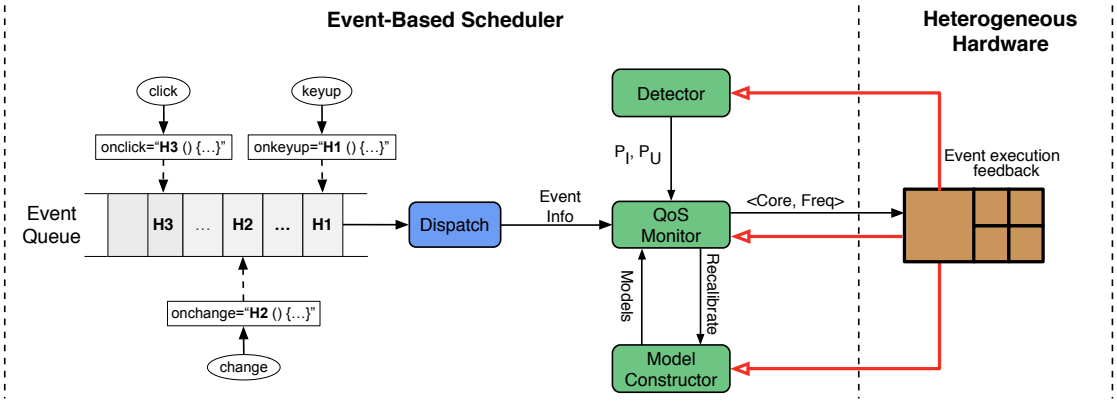

Fig. 10: The EBS runtime framework.

guided QoS hints such as those enabled by the `GreenWeb` language extensions as I will discuss in Sec. 5, the detector can also extract the specified QoS information from the application. The model constructor builds a performance and energy model for each event. The models and event QoS information are then fed into the QoS monitor, which predicts the architecture configuration for executing an event while meeting the specified QoS target.

During application execution, the QoS monitor keeps monitoring event execution time and energy consumption on the hardware and uses the information to adjust its prediction and scheduling decisions on the fly, similar to conventional feedback-driven optimizations [80]. Intuitively, it is possible for the performance and energy models to underpredict or overpredict the architecture configuration. Under such circumstances, the monitor can decide to tune the predicted frequency or transition between big and little cores. If the models are deemed completely unusable, the monitor informs the model constructor to recalibrate the models. We now describe some key implementation details of the QoS monitor operations.

Performance Model The performance model predicts event execution time under different frequencies. We use the classical DVFS analytical model initially proposed in [86], and employed in subsequent work, such as [85]:

$$Execution\ \ time = T_{memory} + N_{dependent} / f$$

where f is the CPU frequency, T_{memory} is the absolute memory access time that does not change with respect to the CPU frequency, and $N_{dependent}$ is the number of CPU cycles that are not overlapped with the memory accesses.

Strictly speaking, $N_{dependent}$ is a function of f. However, precisely constructing a model that varies $N_{dependent}$ with f is complex and introduces a large calibration overhead at runtime. In our experiments, we find that it is feasible and necessary to trade model precision for performance. In particular, we find that treating $N_{dependent}$ as a constant is sufficient in our case. Given this simplification, the model constructor builds the model by profiling twice, one at the maximum frequency and one at the minimum frequency.

Energy Model The energy model predicts the energy consumption of an event execution. We construct the energy model based on the performance model and the estimated power consumption. We derive the power estimation of all the core and frequency combinations by performing a profiling run and storing the results in a local power profile file that is read by the Web application upon every launch. Persisting the power profile file aligns with the Android standard [18].

QoS Monitor's Operation The QoS monitor constructs a deterministic finite automaton (DFA) for each event to keep track of what architectural configuration it needs to provide. The first two times an event is executed, the QoS monitor informs the model constructor to build the performance and energy models. The models let the monitor predict the architecture configuration during all subsequent executions of the event handler.

After the initial model construction, the QoS monitor keeps monitoring event execution in order to perform fine-grained tuning. Specifically, the monitor compares the measured event execution time with the QoS target. The monitor conservatively deems an event's model as overpredicting (or underpredicting) if the measured value is lower than 80% (or higher than 90%) of the QoS target. We empirically adopt these two threshold values because they are found to be effective in practice. Using a two-bit saturating counter, the monitor increases the frequency by 100 MHz or transitions from the little core to the big core if the model is underpredicting, or vice versa.

The monitor switches from fine-tuning an event handler's execution to recalibrating its model if it detects that the model is not performing well. We use a simple heuristic that is efficient in practice. If the model mispredicts (i.e., either underpredicts or overpredicts) more than four consecutive times, the monitor requests the model constructor to recalibrate.

4.4 Related Work

Single ISA Heterogeneous Scheduling The particular implementation of `WebRT` is an example of utilizing single-ISA heterogeneous systems for trading off performance with energy [62]. Nvidia's Kal-El [1] is a single-ISA heterogeneous system that integrates four high-frequency cores with one low-frequency core. ARM's proposed big.LITTLE system [2] contains an out-of-order Cortex-A15 processor and an in-order Cortex-A7 processor. ACMP architecture is already widely adopted in today's mobile SoCs shipped by major vendors. We expect our `WebRT` implementation to be readily applicable to commodity mobile hardware.

`WebRT`'s prediction-based scheduling technique is similar to other recent heterogeneous scheduling proposals, such as PIE [44]. However, instead of relying on (micro)architecture- and system-level statistics for prediction, we capture the complex behavior of webpage characteristics using regression modeling, and accurately predict the webpage load time and energy consumption.

General Software Support for Web Optimizations Most prior research focus on parallelizing browser tasks, such as parsing, CSS selection, etc. [69, 70, 32, 64]. Although such parallelized algorithms can achieve speedups ranging from 4X to 80X for various browsing tasks, they typically do not scale well beyond four cores with the expense of potential energy inefficiency. Thus, while parallelization has potential in desktop systems, it is less favorable for mobile Web computing.

Thiagarajan et al. [82] break the Web browser's energy consumption into coarser-grained elements, such as CSS and Javascript behavior, and identify a few system- and application-level optimizations to improve the energy consumption of mobile Web browsing. The optimizations they recommend, such as reorganizing JavaScript files and removing unnecessary CSS rules, are orthogonal and complementary to our webpage prediction and scheduling work.

Another portion of software-level optimizations focuses on improving the execution model of the Web browser through asynchronous/multiprocess rendering, resource prefetching, smarter browser caching, etc. [67, 68, 89, 23, 3]. All these techniques are orthogonal and can be integrated with my proposal, which primarily focused on the core rendering engine of a Web browser.

5 GreenWeb: Web Language Extensions for Energy-Efficient Web Computing

Web languages are at the interface between applications and Web runtime. Traditionally, Web developers use Web languages to express structure, style, and functionality of an application while relying on the underlying system to perform energy optimizations without compromising user QoS experience. However, without being informed with the QoS information, the runtime system might not always effectively reason about the trade-off between QoS and energy consumption.

To better guide runtime optimizations, I present `GreenWeb`, a set of Web language extensions that allow Web developers to express user QoS expectation at an abstract level. Based on the programmer-guided QoS information, the runtime substrate of `GreenWeb` could then dynamically determine how to deliver the target QoS experience while minimizing the energy consumption.

In this section, I first discuss the relationship between QoS, performance, and energy consumption. It lays the foundation of abstracting user QoS experience (Sec. 5.1). I then introduce QoS type and QoS target as two fundamental abstractions that capture two critical aspects of user QoS experience (Sec. 5.2). I then present the design and specification of `GreenWeb`, a group of Web language extensions that allows developers to easily express the two QoS abstractions (Sec. 5.3). I further discuss the relationship between the `GreenWeb` extensions and the `WebRT` runtime (Sec. 5.4), and argue that `GreenWeb` and `WebRT` are synergistic while independent. Finally, I discuss `GreenWeb` in the general context of language support for performance and energy-efficiency (Sec. 5.5).

5.1 Trade-off Between QoS, Performance, and Energy

We illustrate the relationship between application QoS, performance, and energy savings in Fig. 11. Performance degrades from left to right on the x-axis. The left and right y-axes indicate QoS and energy savings, respectively. Foundational work in human-computer interaction research [47, 79, 38, 71, 72, 73] indicates that interactive application QoS can be classified into three distinct states as machine performance degrades: *imperceptible* $[P_H, P_I]$, *tolerable* $(P_I, P_U]$, and *unusable* $(P_U, P_L]$.

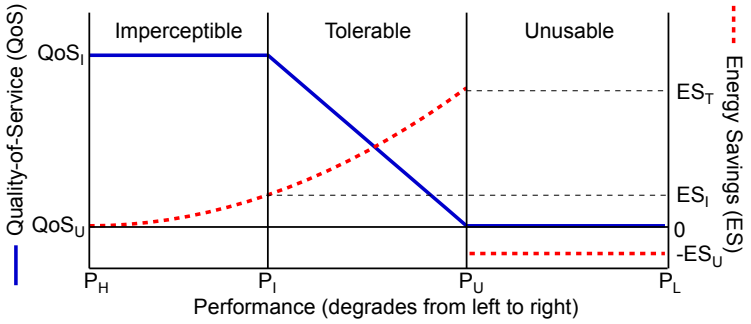

Fig. 11: The interplay between QoS, performance, and energy.

In the imperceptible region, performance can degrade without any user-perceptible QoS loss while achieving more energy savings. Imperceptible QoS, QoS_I, is maintained until performance reaches P_I, the lowest performance level that provides QoS_I. In the imperceptible region, supplying higher performance simply leads to more energy waste without adding any end-user value. For example, the most conservative approach to guarantee application QoS is to supply the peak performance of P_H; it leads to an energy waste of ES_I. Beyond P_I, application QoS enters the tolerable region, where QoS deteriorates as performance reduces, but still remains tolerable. Any QoS could be acceptable in this region depending on the usage scenario or specific user

pattern [87, 90]. Therefore, the tolerable QoS region exhibits a traditional performance-energy trade-off space. As performance further degrades, QoS is eventually violated at P_U, which is the performance limit where users no longer feel engaged by the application. At P_U and beyond, users abandon the service. As a result, any energy consumed up until the service abandonment (ES_U) is wasted, because the underlying computation does not provide any utility to the user.

In summary, QoS-aware energy-efficiency optimization implies one of the following two optimization strategies depending on user QoS expectation. First, when the QoS expectation is high, guarantee imperceptible QoS experience with the minimal energy by exploiting the performance slack between P_H and P_I. Second, when the user QoS expectation is low, guarantee usable QoS experience with the minimal energy by achieving a performance of P_U.

5.2 QoS Abstractions for Web Applications

Abstracting user QoS experience is the prerequisite in expressing QoS information in a programming language. Based on the understanding of the interplay between QoS and energy consumption, I propose the QoS type and QoS target abstractions. I discuss why they are necessary and sufficient to capture user QoS experience for QoS-aware energy-efficiency optimizations.

QoS Type We define an abstraction called *QoS type* to capture different ways that users interpret QoS experience. Two major QoS types exist: *single* and *continuous*. Intuitively, they indicate whether the QoS experience is determined by the "responsiveness" of a single frame or the "smoothness" of a continuous sequence of frames, respectively. Alternatively, one can think of QoS type as the metric of concretizing performance (x-axis) in Fig. 11.

- Some user interactions produce only a single frame, which we call the response frame. The QoS type of these interactions is "single," indicating that user QoS experience is determined by the latency at which the response frame is perceived by users [47]. For instance, imagine a fingertap interaction that opens a search box in a Web application. Users perceive the effect of the fingertap when the application displays a response frame—the frame with the search box displayed. Web application loading process also falls in this category. This is because although there are several intermediate frames being produced during the loading process, user QoS experience is largely determined by the latency to deliver the "first meaningful frame" [76], which indicates that a Web application is usable to users.
- The other QoS type is "continuous," corresponding to interactions whose responses are not one single frame but a sequence of continuous frames. User QoS experience is determined by the latency of *each* frame in the sequence, rather than one particular frame.

  Continuous frames are often found in the form of animations. The simplest form of animation is triggered by finger moving such as scrolling. Tapping can also cause a sequence of frames to be generated. For instance, many Web applications provide a navigation button that dynamically expands when tapped and generates an animation. More complex animations in Web applications can be controlled by `requestAnimationFrame` (`rAF`) APIs [22] and CSS animation/transition [8, 10].

QoS type is an important abstraction because optimizing for a wrong QoS type may lead to energy-inefficient decisions. For example, imagine a fingtertap that only produces one frame (e.g., displaying a search box). A Web runtime that mistakenly optimizes for "continuous" frame latency may force the hardware to run at its peak performance in order to produce frames even

Table 3: Mobile Web interactions fall into three categories based on different QoS type and QoS target combinations.

QoS Type	QoS Target (P_I, P_U)	Description
Continuous	(16.6, 33.3) ms	QoS experience is evaluated by *continuous* frame latencies.
Single	(100, 200) ms	QoS experience is evaluated by a *single* frame latency. Users expect *short* response period.
	(1, 10) s	QoS experience is evaluated by a *single* frame latency. Users expect *long* response period.

when no frame is intended, leading to energy waste. "Single" and "continuous" are two unique QoS types that must be differentiated.

QoS Target Another critical QoS abstraction is *QoS target*, denoting the performance level needed to deliver a certain QoS experience. Two different QoS targets exist that are critical to user experience: imperceptible target (P_I) and usable target (P_U) [91]. They correspond to the imperceptible and usable QoS experience discussed in Fig. 11. We use frame latency as a natural choice for the performance metric because frame updates dictate QoS experience. Specifically, we define frame latency as the delay from when an event is initiated by a user to when its corresponding frame(s) show on the display.

- For interactions with a "single" QoS type, QoS target depends on the complexity of the interaction [47]. For interactions that are expected to finish quickly, user latency tolerance is low. For instance, a fingertap that displays a search box falls into this category because displaying a search box is inherently expected to finish "instantly." For these "lightweight" interactions, users feel the system is responding instantly at 100 ms, and start thinking that the system is not working after 300 ms [20]. Thus, 100 ms and 300 ms can be used as the P_I and P_U values, respectively.

  In contrast, when users are aware of a computationally intensive job being processed, they tend to have high tolerance for latencies [73]. Psychology studies show that users can subconsciously wait up to 1 second for a job to complete while still staying focused on the current train of thought. Once a job execution exceeds 10 seconds, user attentions are distracted and cannot tolerate the delay [38, 71]. Therefore, 1 second and 10 seconds can be treated as the P_I and P_U values for "heavyweight" interactions, respectively.

- For interactions with a "continuous" QoS type, 60 and 30 frames per second (FPS) deliver a "seamless" and "just playable" user experience, respectively [43]. Thus, a performance level that guarantees 16.6 ms and 33.3 ms frame latency can be regarded as the imperceptible and usable QoS target, respectively. It is worth noting that the QoS target applies to each frame rather than an average latency. This is because human eyes are very sensitive to frame variance. Tiny hitches in a high volume of frames can cause a poor QoS experience and even headaches [14, 16].

Table 4: `GreenWeb` API specification. Each API is a CSS rule specifying the QoS information when a particular event is triggered on a Web element.

Syntax	Semantics
`E:QoS { onevent-qos: continuous; }`	As soon as `onevent` is triggered on DOM element `E`, the application must continuously optimize for frame latency. Use the P_I and P_U values in Table 3 as the default QoS target for all frames.
`E:QoS { onevent-qos: single, short| long }`	Once `onevent` is triggered on element `E`, the application must optimize for the latency of the single frame caused by `onevent`. Users expext short (longer) latency. Use the P_I and P_U values in Table 3 as the default QoS target.
`E:QoS { onevent-qos: continuous| single, ti-value, tu-value, }`	Explicitly specify P_I (`ti-value`) and P_U values (`tu-value`) for QoS targets. Note that both values must either appear or be ommitted together.

User interactions fall into three distinct categories based on different QoS type and QoS target combinations as listed in Table 3. Although the absolute values of QoS target (P_I and P_U) in each category can vary slightly with user perceptibility, their magnitudes differ significantly across categories (i.e., tens of milliseconds versus hundreds of milliseconds versus seconds). Thus, QoS target is an important abstraction to differentiate different performance requirements.

5.3 QoS-Aware Web API Design

We present `GreenWeb`, a set of Web language extensions that lets application developers easily express the two QoS abstractions as program annotations. `GreenWeb` APIs extend current CSS language to specify QoS type and QoS target information. We choose CSS because its syntax and semantics naturally allow us to select DOM elements and specify various characteristics. The core of CSS is a set of *style rules* [7]. Each style rule selects specific Web application elements and sets their style properties. A style rule expresses such semantics through two language constructs: a *selector*, which selects specific Web application elements, and a set of style *declarations*, which are $\langle property, value \rangle$ pairs that assign *value* to *property*. As an example, the following CSS rule `h1 {font-weight: bold}` selects all the `h1` elements and sets their `font-weight` property to `bold`.

Table 4 lists semantics of each `GreenWeb` API. Intuitively, each `GreenWeb` API selects an application element `E`, and declares CSS properties to express the QoS type and QoS target information when an event `onevent` is triggered on `E`. We now describe the details of `GreenWeb` extensions.

Selector To decorate a CSS rule as specifying QoS information of an element, we define a new CSS pseudo-class selector [9] "`:QoS`." An element `E` is selected using existing selectors, such as ID (`#id`) and Class (`.class`) selectors, before applying the `:QoS` pseudo-class qualifier. For example, `div#intro:QoS` selects the `div` element with the ID `intro` before declaring QoS information.

Property QoS information is expressed as CSS properties in GreenWeb. We define a new CSS property called onevent-qos, in which onevent is a DOM event that GreenWeb supports. In its simplest form, onevent-qos could be set to continuous (first rule in Table 4). The semantics of declaring onevent-qos: continuous is that as soon as onevent is triggered on element E, the runtime must continuously optimize for frame latency until the last relevant frame is generated.

To express the "single" QoS type, the onevent-qos property accepts a list of two values separated by a comma, one to indicate that the QoS type is single, and the other to indicate whether users expect a short or long execution period (second rule in Table 4). For instance, the declaration onevent-qos: single, short expresses that the runtime must optimize for the latency of the single frame caused by onevent, and users expect short frame latency.

Developers do not have to specify the QoS target values, and the GreenWeb runtime will use the P_I and P_U values in Table 3 as the default QoS target. However, we also provide the flexibility for developers to overwrite default QoS targets. This is achieved by specifying absolute values of P_I and P_U (in milliseconds) after single or continuous, as illustrated by the third rule in Table 4.

5.4 GreenWeb and WebRT Inteplay

Combing GreenWeb and WebRT in an integrated system would provide programmers an opportunity to guide runtime energy-efficiency optimizations by providing QoS "hints." This approach would be both *precise* and *efficient*. It is precise because only developers have the exact knowledge of code logic. They can provide QoS type and target information that is difficult for the runtime to infer. It is efficient because it does not entail performance and energy overhead of runtime detection. Such a design philosophy is similar to traditional pragma-based programming APIs such as OpenMP. For example, the "omp for" pragma in OpenMP indicates that iterations in a for loop are completely independent such that the runtime can safely parallelize the loop without the need to check for correctness. Similarly, GreenWeb annotations would allow the Web runtime to perform "best-effort" optimizations without having to infer QoS information. I leave the full integration and evaluation of such a language-runtime co-designed system as future work.

It is worth noting that although GreenWeb and WebRT are compatible and can be integrated, they by design are independent systems. GreenWeb is independent of WebRT because it does not pose constraints on specific runtime implementations. It is possible to use the ACMP-based WebRT as a GreenWeb runtime implementation to make QoS-energy trade-offs at the hardware level. It is also feasible to build a runtime leveraging only a single big (or little) core capable of DVFS [66, 26]. In addition, one could implement a GreenWeb runtime using pure software-level techniques, such as prioritizing resource loading [36] or using power-conserving colors [46].

WebRT is also independent of GreenWeb because it by design does not rely on programmer-assisted annotations to perform scheduling. Without QoS "hints" from the application, the WebRT schedulers can assume a default QoS target and QoS type for each Loading, Touching, and Moving interaction. The WebRT schedulers can also attempt to infer the QoS type and QoS target by profiling event executions to identify which category in Table 3 that an event falls into. I leave the quantitative comparison between unguided and GreenWeb-guided WebRT to future work.

5.5 Related Work

Language Support for Web Performance The Web community has a long tradition of providing language extensions that allow developers to specify "hints" for browsers. The focus, however,

has been primarily on *performance* optimizations. `GreenWeb`, to the best of our knowledge, is the first Web language extension that specifically targets *energy*.

The most classical example of performance hint is link prefetch [49], which lets Web developers use an HTML tag to specify that a particular link will likely be fetched in the near future. With such information, a Web browser could prefetch the link when there are no on-demand network requests. Another example is the CSS `willChange` property [11], which hints browsers about what visual changes to expect from an element so that the browser could perform a computationally intensive task ahead of time. Similar to `willChange`, `GreenWeb` introduces a new CSS property `onevent-qos`, which allows providing QoS-related hints.

Language Support for Energy Efficiency Language support for energy-efficiency has recently become an important research thrust. Most work targets approximate computing and sensor-based applications. To the best of our knowledge, this is the first work that focuses on Web applications. Green [33] provides APIs that let developers specify approximate versions of a function and QoS loss constraints. EnerJ [77] takes the language support for approximate computing a step further by designing a general type system. These language frameworks could be applicable if sources of approximation in Web applications are identified, which however is beyond our scope. LAB [59] identifies latency, accuracy, and battery as fundamental abstractions for improving energy-efficiency in sensor-based applications. Similarly, `GreenWeb` identifies the QoS type and QoS target abstractions for enabling energy-efficient Web applications.

6 Retrospective and Prospective Remarks

Retrospective As a promising first step, my proposal explores the feasibility of a holistic system design to improve the energy-efficiency of mobile Web computing while delivering user satisfaction. It argues that the traditional interfaces across the Web computing stack should be enhanced with new abstractions for QoS and hardware. It demonstrates three general principles:

- Empowering Developers Web developers today must be conscious of energy efficiency because of increasing user awareness. Current application/runtime abstractions, however, do not provide developers opportunities to optimize for energy efficiency. Pure runtime-based techniques are QoS-agnostic. A key principle in my proposal is that developers should be integrated into the energy optimization loop by empowering them to express user QoS expectations at an abstract level. `GreenWeb` demonstrates the first step.
- Exposing Hardware Complexity Mobile system-on-chip (SoC) designs are undergoing rapid design iteration and innovation with each generation incorporating more complex cores and accelerators. Future Web browser runtimes must embrace the unprecedented hardware complexity. To enable that, traditional runtime/architecture interfaces must be enhanced to expose a certain level of hardware details. `WebRT` demonstrates the potential of exposing CPU core type and frequency and designing the Web runtime accordingly.
- General-Purpose vs. Specialization Ultimately future mobile processor designs have to improve both the performance and energy-efficiency simultaneously, not just making trade-offs between the two design goals. Architectural specialization comes as a first choice to achieve both improvements. However, I argue that retaining the general-purpose programmability during the specialization process is of critical importance for the Web stack because of its inherent complexity. `WebCore` starts from a (well-customized) general-purpose design and incorporates modest specialization for the most lucrative target.

Prospective I discuss three key steps in generalizing the principles and improving the proposal.

- Easing Developers' Effort Although developers must be empowered, they should not be over-burdened. A key limitation of the current `GreenWeb` proposal is that it requires developers to manually annotate Web applications with QoS information. To make `GreenWeb` more practical, I will explore the feasibility of automatically applying `GreenWeb` annotations.

  The rationale for designing an automatic annotation system is twofold. First, some Web developers may not want to spend the extra effort of manual annotation, such as for legacy applications whose code logic might be obscure. Second, in complex Web applications with many DOM events, manually going through all events and identifying their QoS type and QoS target could be cumbersome. In both cases, an automated system would automatically find all the events and annotates them with the two QoS abstractions, enabling QoS-aware energy-efficiency optimizations without developers' intervention.

  I will also evaluate the trade-off between automatic and manual annotation. Specifically, I will use three metrics: coverage, accuracy, and effort. *Coverage* is measured by the percentage of events that can be annotated. *Accuracy* is assessed by whether an event's QoS type and QoS target information can be correctly annotated. *Effort* is measured by the amount of time required to perform annotation. Intuitively, the manual annotation approach can achieve 100% coverage at 100% accuracy–after all, developers have all the knowledge of the exact application logic, theoretically. However, manual annotation would also lead to

significant annotation effort, especially for production-scale Web applications. On the other hand, the automatic annotation approach would require much less annotation effort with the caveat of not being able to reach a perfect accuracy and coverage. My future work will quantify such a comparison and make a step further toward pushing `GreenWeb` to practice.

- Composability of GreenWeb There are two aspects of composability that need to be addressed: program-level and abstraction-level. At the program level, an ideal `GreenWeb`-annotated program can be integrated with other non-annotated code while ensuring functionality (correctness). After all, `GreenWeb` extensions concern with QoS and traditional language constructs concern with functionality (correctness). This composability ensures a *separation of concerns* between QoS and functionality (correctness) in Web programming.

  The program-level composability also allows developers to annotate applications that make use of close-sourced libraries. For example, even if an event callback is implemented using a third-party animation library API whose source code is not available, developers should still be able to specify the event's QoS type as "continuous" as long as the API is guaranteed to produce an animation. This capability allows QoS annotations to remain largely unchanged even when a library's actual implementation evolves over time.

  In my planned work toward thesis defense, I will evaluate the program-level composability as it is a short-term concern that affects the usability of `GreenWeb` extensions. The key is to make sure that the runtime implementation of QoS annotations is "sandboxed" and does not leak information to the "functionality world." In this way, even when an QoS annotation is erroneous the application's functionality and correctness will not be interfered with.

  The other aspect of composability is at the QoS abstraction level. It is not clear if one can easily express a new QoS abstraction based on the two proposed QoS abstractions, QoS type and QoS target. The next step of Web language research should understand how to design QoS "primitives," based on which complex, higher-level QoS abstractions can be easily composed. This composability is critical in the long term when we will see new user interaction forms and new ways that users assess QoS experience. Enumerating every single type of QoS in a programming model is not scalable. With composability at the QoS abstraction level, however, a Web programming model could simply provide a library that lets developers construct different QoS specifications in a completely customized way.

  I will leave the abstraction-level composability for the next phase of research because it is a long-term concern that involves extensively surveying future human-computer interaction forms and new QoS specifications. I believe that the QoS type and QoS target abstractions are sufficient for expressing predominant QoS specifications on today's mobile devices.

- Self-adaptive Web Runtime The Web is becoming a universal application development platform in the Internet-of-Things era. The underlying Web runtime is facing a unique challenge of operating on different device capabilities and in vastly different environments with different QoS, energy, and reliability constraints. The future Web runtime must be self-adaptive. As a first step, I will conduct two research items. First, I will evaluate the sensitivity of `WebRT` with respect to different QoS targets. For example, I will understand can the webpage-aware scheduler achieve the same level of energy savings when the cut-off latency is relaxed. Second, I will evaluate the sensitivity of `WebRT` on mobile devices with different capabilities. For instance, I will evaluate `WebRT` on an ARM Cortex A7 processor to mimic a smartwatch-level processor (as opposed to a smartphone).

Timeline Toward Thesis Defense

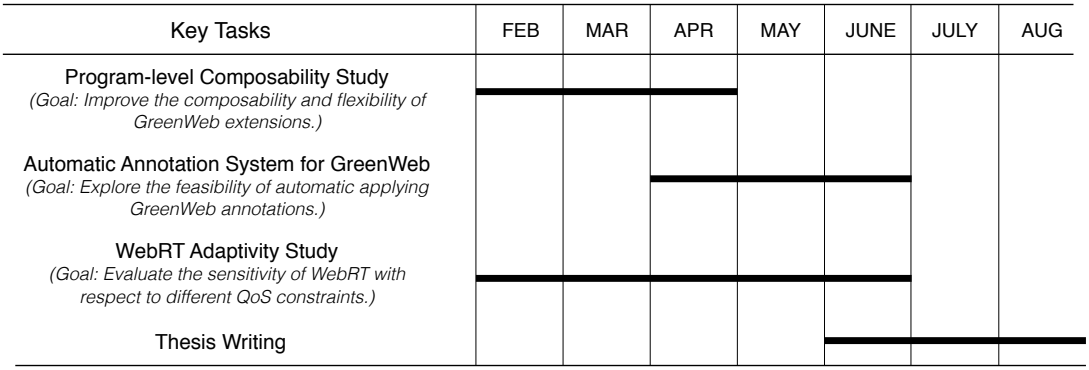

Fig. 12: A plan of action for future research and dissertation writing.

Publications Applicable to the Proposal

[**PLDI 2016**] Yuhao Zhu and Vijay Janapa Reddi, "GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing."

[**HPCA 2016**] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction."

[**MICRO 2015**] Yuhao Zhu, Daniel Richins, Matthew Halpern and Vijay Janapa Reddi, "Server-Side Microarchitectural Implications of Asynchronous Event-Driven Web Applications."

[**HPCA 2015**] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. "Event-based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications."

[**ISCA 2014**] Yuhao Zhu and Vijay Janapa Reddi, "WebCore: Architectural Support for Mobile Web Browsing."

[**HPCA 2013**] Yuhao Zhu and Vijay Janapa Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems."

[**ISPASS 2015**] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. "Mosaic: Cross-platform User-interaction Record and Replay for the Fragmented Android Ecosystem."

[**IEEE MICRO 2015**] Yuhao Zhu and Vijay Janapa Reddi, "The Role of the CPU in Energy-Efficient Mobile Web Browsing."

[**CAL 2014**] Yuhao Zhu, Aditya Srikanth, Jingwen Leng and Vijay Janapa Reddi, "Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing."

References

[1] NVidia: Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. http://goo.gl/uWZJ1

[2] Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. http://goo.gl/7mgbL

[3] Mozilla: "Speculative parsing in firefox". http://goo.gl/tJlRk

[4] "3G/4G Wireless Network Latency: How did Verizon, AT&T, Sprint and T-Mobile Compare in July 2014?" http://www.fiercewireless.com/special-reports/3g4g-wireless-network-latency-how-did-verizon-att-sprint-and-t-mobile-compa-3

[5] "Are smartphones getting larger because they have to?" http://www.extremetech.com/computing/163636-the-ever-expanding-smartphone-or-why-are-phablets-so-darn-popular

[6] "big.little technology: The future of mobile." https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[7] "Cascading style sheets level 2 revision 1 (css 2.1) specification." http://www.w3.org/TR/CSS2/

[8] "CSS Animations." http://www.w3.org/TR/css3-animations/

[9] "CSS Pseudo-classes." http://www.w3.org/TR/selectors/#pseudo-classes

[10] "CSS Transitions." http://www.w3.org/TR/css3-transitions/

[11] "Css will change module level 1." http://www.w3.org/TR/css-will-change-1/

[12] "GPU Accelerated Compositing in Chrome." www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

[13] "Heterogeneous multi-processing solution of exynos 5 octa with arm big.little technology." https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[14] "Jank busting for better rendering performance." http://www.html5rocks.com/en/tutorials/speed/rendering/

[15] "LTE Subscriptions to Surpass 1 Billion This Year." http://www.pcworld.com/article/2935332/lte-subscriptions-to-surpass-1-billion-this-year.html

[16] "Nvidia: Adaptive vsync technology." http://www.geforce.com/hardware/technology/adaptive-vsync/technology

[17] "OOKLA Speedtest for Android." http://www.speedtest.net/mobile/android/

[18] "Power profiles for android." https://source.android.com/devices/tech/power.html#

[19] "Smartphone screen sizes keep on growing—but not for much longer." http://www.wired.com/2013/04/why-big-smartphone-screens/

[20] "Speed, performance, and human perception." http://chimera.labs.oreilly.com/books/1230000000545/ch10.html#SPEED_PERFORMANCE_HUMAN_PERCEPTION

[21] "SunSpider Benchmark." https://www.webkit.org/perf/sunspider/sunspider.html

[22] "Timing Control for Script-based Animations." http://www.w3.org/TR/animation-timing/

[23] "Webkit2." http://trac.webkit.org/wiki/WebKit2

[24] "System software for ARM big.LITTLE systems," in *ARM Whiltepaper*, 2011.

[25] "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," in *ARM Whiltepaper*, 2013.

[26] "Nvidia tegra 4 family cpu architecture: 4-plus-1 quad core," in *Nvidia Whiltepaper*, 2013.

[27] 7-cpu, "ARM Cortex-A15 Specification," 2015. http://www.7-cpu.com/cpu/Cortex-A15.html

[28] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *Proc. of ISCA*, 2000.

[29] Alexa, "Alexa," 2015. http://www.alexa.com/

[30] ARM, "Exploring the Design of the Cortex-A15 Processor," 2012. http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf

[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] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Proc. of PLDI*, 2010.

[34] BatteryUniversity, "Battery statistics," 2011. http://batteryuniversity.com/learn/article/battery_statistics

[35] 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.

[36] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices," in *Proc. of NSDI*, 2015.

[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] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The Information Visualizer: An Information Workspace," in *CHI*, 1991.

[39] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proc. of USENIX ATC*, 2010.

[40] G. Chadha, S. Mahlke, and S. Narayanasamy, "EFetch: Optimizing Instruction Fetch for Event-driven Web Applications," in *Proc. of PACT*, 2014.

[41] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating Asynchronous Programs through Event Sneak Peek," in *Proc. of ISCA*, 2015.

[42] X. Chen, Y. Chen, Z. Ma, and F. C. Fernandes, "How Is Energy Consumed In Smartphone

Display Applications?" in *Proc. of HotMobile*, 2013.

[43] M. Claypool, K. Claypool, and F. Damaa, "The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games," in *Proc. of Multimedia Computing and Networking*, 2006.

[44] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-cores through Performance Impact Estimation (PIE)," in *Proc. of ISCA*, 2012.

[45] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc., "Design of Ion–Implanted MOSFET's With Very Small Physical Dimensions," in *IEEE Journal of Solid-State Circuits*, 1974.

[46] M. Dong and L. Zhong, "Chameleon: a color-adaptive web browser for mobile oled displays," in *Proc. of MobiSys*, 2012.

[47] Y. Endo, Z. Wang, J. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," in *Proc. of OSDI*, 1996.

[48] D. Evans, "The Internet of Things: How the Next Evolution of the Internet is Changing Everything," 2011. http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[49] D. Fisher and G. Saksena, "Link prefetching in mozilla: A server-driven approach," in *Web content caching and distribution*. Springer, 2004, pp. 283–291.

[50] I. Grigorik, *High Performance Browser Networking*. O'Reilly, 2013.

[51] 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.

[52] 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.

[53] M. Halpern, Y. Zhu, and V. J. Reddi, "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*, 2016.

[54] 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.

[55] F. E. Harrell, *Regression Modeling Strategies*. Springer, 2001.

[56] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer, 2009.

[57] U. Hoelzle, "The Google Gospel of Speed," 2012. https://www.thinkwithgoogle.com/articles/the-google-gospel-of-speed-urs-hoelzle.html

[58] O. Hub, "Chromium Project Summary: Languages." 2015. https://www.openhub.net/p/chrome/analyses/latest/languages_summary

[59] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing," in *Proc. of OOPSLA*, 2013.

[60] Kissmetrics, "How Loading Time Affects Your Bottom Line," 2011. https://blog.kissmetrics.com/loading-time/

[61] KPCB, "2015 Internet Trends," 2015. http://www.kpcb.com/blog/2015-internet-trends

[62] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," in *Proc. of MICRO*, 2003.

[63] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *Proc. of ASPLOS*, 2006.

[64] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the Efficiency of Text Processing on Commodity Processors," in *Proc. of HPCA*, 2012.

[65] 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.

[66] D. Lo, T. Song, and G. E. Suh, "Prediction-guided performance-energy trade-off for interactive applications," in *Proc. of MICRO*, 2015.

[67] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "PocketWeb: Instant Web Browsing for Mobile Devices," in *Proc. of ASPLOS*, 2012.

[68] H. Mai, S. Tang, S. T. King, C. Cascaval, and M. Pablo, "A Case for Parallelizing Web Pages," in *Proc. of USENIX HotPar*, 2012.

[69] L. A. Meyerovich and R. Bodik, "Fast and Parallel Webpage Layout," in *Proc. of WWW*, 2010.

[70] L. A. Meyerovich and R. Bodik, "FTL: Synthesizing a Parallel Layout Engine," in *Proc. of ECOOP*, 2012.

[71] R. B. Miller, "Response Time in Man-computer Conversational Transactions," in *AFIPS Fall Joint Computer Conference*, 1968.

[72] B. A. Myers, "The Importance of Percent-done Progress Indicators for Computer-human Interfaces," in *Proc. of CHI*, 1985.

[73] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.

[74] OpenSignal, "Android Fragmentation Visualized," 2014. http://opensignal.com/assets/pdf/reports/2014_08_fragmentation_report.pdf

[75] 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.

[76] K. Sakamoto, "Time-to-first-x-paint metrics: Status and refinement plans," 2015. https://docs.google.com/document/d/1Owfs6arciEnWgT2-8bWCcHdYRIKRKZ0Xj8UtqRx4c3k/edit

[77] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proc. of PLDI*, 2011.

[78] F. Schlachter, "No moore's law for batteries," in *Proc. of National Academy of Science of the United States of America*, 2013.

[79] B. Shneiderman, *Designing the User Interface*. Addison-Wesley, 1992.

[80] M. D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization (Keynote Talk)," in *Proc. of DYNAMO*, 2000.

[81] M. A. Suleman, Y. N. Patt, E. Sprangle, A. Rohillah, A. Ghuloum, and D. Carmean, "Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency," The University of Texas as Austin, Technical Report TR-HPS-2007-001, 2007.

[82] 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.

[83] W3C, "CSS Cascading Order," 2014. http://www.w3.org/TR/CSS2/cascade.html#cascade

[84] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "AnySP: Anytime Anywhere Anyway Signal Processing," in *Proc. of ISCA*, 2009.

[85] Q. Wu, V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Proc. of MICRO*, 2005.

[86] F. Xie, M. Martonosi, and S. Malik, "Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits," in *Proc. of PLDI*, 2003.

[87] Y. Xu, M. Lin, H. Lu, G. Cardone, N. D. Lane, Z. Chen, A. T. Campbell, and T. Choudhury, "Preference, Context and Communities: A Multi-faceted Approach to Predicting Smartphone App Usage Patterns," in *Proc. of ISWC*, 2013.

[88] A. Yogasingam, "Teardown: Samsung galaxy s4," 2013. http://www.edn.com/design/power-management/4412960/5/Teardown--Samsung-Galaxy-S4-

[89] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart Caching for Web Browsers," in *Proc. of WWW*, 2010.

[90] Z. Zhao, M. Zhou, and X. Shen, "SatScore: Uncovering and Avoiding a Principled Pitfall in Responsiveness Measurements of App Launches," in *Proc. of UbiComp*, 2014.

[91] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications," in *Proc. of HPCA*, 2015.

[92] Y. Zhu, M. Halpern, and V. J. Reddi, "The Role of the CPU in Energy-Efficient Mobile Web Browsing," in *Micro, IEEE*, 2015.

[93] Y. Zhu and V. J. Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems," in *Proc. of HPCA*, 2013.

[94] Y. Zhu and V. J. Reddi, "WebCore: Architectural Support for Mobile Web Browsing," in *Proc. of ISCA*, 2014.

[95] Y. Zhu and V. J. Reddi, "GreenWeb: Language Extensions for QoS-aware Energy-Efficient Mobile Web Computing," in *Conditionally accepted to PLDI*, 2016.

[96] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Server-Side Microarchitectural Implications of Asynchronous Event-Driven Web Applications," in *Proc. of MICRO*, 2015.

[97] Y. Zhu, A. Srikanth, J. Leng, and V. J. Reddi, "Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing," in *Computer Architecture Letters*, 2014.