The Dissertation Committee for Yuhao Zhu
certifies that this is the approved version of the following dissertation:

Energy-Efficient Mobile Web Computing

Committee:

____
Vijay Janapa Reddi, Supervisor

____
Lizy K. John

____
Derek Chiou

____
Christine Julien

____
Scott Mahlke

Energy-Efficient Mobile Web Computing

by

Yuhao Zhu, B.S.,; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to my wife Yunjing Luo

and my loving parents, Qinxiong Zhu and Xiaojing Yu

Acknowledgments

I wish to thank the multitudes of people who helped me throughout this journey. First and foremost, I would like to thank The University of Texas at Austin, and the taxpayers of the great state of Texas, for supporting me to pursue a degree in a filed that I am truly passionate about. Among many things, UT Austin provides a stimulating, diverse, and welcoming environment for intellectual explorations that one could ever imagine having.

I would like to express my sincere gratitude to my advisor Vijay Janapa Reddi, who turned me from a student to a researcher. I came to UT as an avocational computer architecture enthusiast. Vijay patiently taught me how to turn half-baked ideas to high-quality research papers, and how to turn individual papers into a dissertation. I appreciate the many technical and visionary discussions that he had with me while walking on campus.

Vijay also trained me to be a long-term investor. In my first semester working with Vijay, he spent over a thousand dollars to send us to a half-day course on information visualization. He later sent us to other venues such as training mental agility. Although some of these investments might have not paid off immediately toward a paper—some never will—it gradually shaped my mindset beyond just becoming a better graduate student to becoming a better person, which I will forever benefit from.

I would also like to thank other members of my dissertation committee: Lizy John, Derek Chiou, Christine Julien, and Scott Mahlke. Their keen insights and feedbacks are absolutely invaluable in developing and refining the concepts in my dissertation. In addition, Lizy taught me Computer Performance Evaluation and Benchmarking, and Derek taught me Parallel Architecture; both helped me form the computer architecture foundations. Special thanks to Christine and Scott who gave me many pieces of advice and wrote me recommendation letters for my job search.

During my time at UT, I also had the privilege to interact and work with other faculty members, in particular Mattan Erez, Mohit Tiwari, and Yale Patt. Mattan is a calm, sharp, and encouraging human being who I look up to. He taught me Computer Architecture during my first semester in UT (and this country), which is the reason that I now can talk comfortably about computer architecture. Mohit is always there to provide me lots of advice in career development. I worked with Dr. Patt during my first year as a graduate researcher and as a teaching assistant for his Computer Architecture course in my second semester. The interactions with him provided me unique perspectives that I would have gotten from nowhere otherwise.

I had four summer internships during my graduate study, and I want to thank my mentors during each internship. Osvaldo Colavin was my mentor at STMicroelectronics in summer 2011. We first met at DAC that year at San Diego, and he later made me an internship offer. I thank Osvaldo for truly opening my eyes to the industry for the first time. Mauricio Breternitz was my mentor during my two internships at AMD Research in summer 2012 and 2013. I admire his incredible experience in hardware and software industry. He gave me the opportunity to integrate several AMD specific hardware performance counters into the `libpfm` in my first internship, and gave me lots of freedom to explore Web browser caching during my second internship. Nat Duca was my mentor during my internship at the Google Chrome team in summer 2015. Even to this day I couldn't believe that I had worked with Nat. Nat is an incredible person and great friend who continues to help me after the internship. He also connected me with many other wonderful Chrome engineers—the reason that the Web is well alive and kicking!

My lab mates and friends at UT Austin not only help me academically, but also always encourage me to never back off from challenges, and I greatly thank them for that. Jingwen Leng and Aditya Srikanth helped me with my first project to get my research started. Jingwen and I had a lot of fun beyond research. Among many things, we often drove to Houston just to play snooker, and flew to UK to watch snooker games. Daniel Richins and Wenzhi Cui helped me with the Node.js project. Wenzhi would always listen to my nonsense ideas. Matthew Halpern helped me with *everything* since he joined. He acts as a cheerleader when I am down and keeps me sane. I have also met many friends at UT and Austin who I want to thank: Song Zhang, Haishan Zhu, Tianhao Zheng, Yazhou Zu, Yong Li, Minsoo Rhu, Milad Hashemi, Faruk Guvenilir, and Khubaib. Special thanks to Khubaib who mentored me during my first year working with Dr. Patt and continued to support me after that.

I am grateful to the Google Ph.D. fellowship for supporting my Ph.D. research. Matt Welsh is my fellowship mentor, who is a true enemy of hogwash. Matt has supported my research throughout my entire graduate study by being an advocator inside Google for our research. He also generously wrote a recommendation letter for my job search. I was also supported by the Microelectronics and Computer Development Fellowship at the Cockrell School of Engineering in UT Austin during my second year.

I also want to thank my undergraduate advisor Yangdong (Steve) Deng at Tsinghua University. I was an undergrad at Beihang University, and Steve just came back from the United States to start his lab. I anxiously sent him a cold email expressing my interests, and (not) surprisingly he decided to bring me to the lab. We together worked on GPGPU programming and microarchitecture, which greatly raised my interests in system architectures. He encourages me to think big and makes me believe that I, too, could publish first-class research results. I would never get a Ph.D. if not for the one year working and interacting with Steve.

Most importantly, not a single word of this dissertation would happen without the support of my loving family. My parents, Qinxiong Zhu and Xiaojing Yu, care about me more than themselves and never hold me off in pursuing whatever I am passionate about, even if that means they only saw me four times in the past seven years. Finally, I thank my wife Yunjing Luo for always being there since we first met in 2012. Although she believes that I could have finished Ph.D. sooner without her, I believe otherwise.

Energy-Efficient Mobile Web Computing

Publication No. ____________________

Yuhao Zhu, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Vijay Janapa Reddi

Next-generation Web services will be primarily accessed through mobile devices. However, mobile devices are low-performance and stringently energy-constrained. In my dissertation, I propose the design of a high-performance and energy-efficient mobile Web 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 is 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 contributions made in this dissertation will likely have a long-term impact because the target application domain, the Web, is becoming a universal mobile development platform, and because our solutions target the fundamental computation layers of the Web domain.

Table of Contents

List of Tables

List of Figures

Chapter 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 [91]. The trend is clear: next-generation Web services will be primarily accessed through mobile devices instead of desktops and laptops as in previous generations [121].

While there are significant growth opportunities for mobile Web computing, standing in the way are technology challenges. Specifically, there is a fundamental tension between the ever-increasing computation intensity of Web technologies and the performance and energy-constrained nature of mobile devices. Such a mismatch between computational "demand and supply" leads to poor quality-of-service (QoS) experience, resulting in severe consequences. For example, Google estimated that "a 400 ms delay leads to a 0.44% drop in search volume.” [110] Similarly, Amazon concluded that a 1-second delay in webpage load time could translate to $1.6 billion lost in sales annually [87].

Conventional techniques to improve mobile compute capability have largely been focused on CPU design, largely by adopting desktop-oriented design techniques, i.e., to apply aggressive (micro-)architecture mechanisms for both single-core and multi-core performance while relying the underlying circuit techniques (i.e., Dennard Scaling [84]) for power and energy-efficiency [104]. However, as the demise of the graceful Dennard scaling becomes a reality [90], excessive power and energy consumption will eventually put the CPU-centric, desktop-like design strategy to its end.

Thesis Statement To sustain mobile performance improvement while being energy-efficient, we must deviate away from the traditional CPU-only mentality. Instead, we must expand the research scope to the entire Web computing stack spanning architecture, runtime, and programming language layers. Following this tenet, my dissertation proposes hardware accelerators, runtime scheduling mechanisms, and programmers-assisted language annotations, the combination of which forms a hardware/software co-designed computing substrate that improves mobile Web performance and energy-efficiency.

The rest of this chapter is organized as follows. Chapter 1.1 provides an overview of my research contributions. Chapter 1.2 discusses the long-term impact of my work. Chapter 1.3 puts my work in the broad context of mobile computing. Chapter 1.4 outlines the rest of the dissertation and Chapter 1.5 lists previously published materials that this dissertation draws upon.

1.1 Research Contributions

The key theme of my work is to deviate away from the general-purpose CPU-centric mindset and to take a holistic view of the mobile Web computation stack spanning application, Web browser runtime, and processor architecture layers. I contend that improving energy-efficiency and performance of mobile Web computing requires us to enhance the traditional computing interfaces with *new abstractions* and to leverage the new interfaces for *cross-layer optimizations*. As such, the central challenge of my research is to carefully forge new abstractions that expose optimization opportunities while enabling effective and practical optimization mechanisms.

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 simplistic and monolithic execution model of the hardware. However, today's mobile processors are becoming extremely complex, combining general-purpose cores that have different performance and energy characteristics [124] with special-purpose domain-specific accelerators. 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 both needs on specializing the processor architecture for the Web domain that enriches the runtime/architecture interface and on designing an intelligent Web browser runtime that can effectively manage the complex interface to optimize for energy-efficiency.

In the spirit of enhancing the traditional Web computing stack interfaces and leveraging the new interfaces to optimize each layer, my dissertation makes the following three contributions. Figure 1.1 provides an overview of the contributions. Enhancements to the existing Web stack are shaded.

- **Web Language Extensions:** I propose `GreenWeb`, a set of programming language extensions that let Web developers express user QoS expectations as program annotations. `GreenWeb` enhances the traditional application-runtime interface with two new programming abstractions, QoS type and QoS target, that capture two critical aspects of user QoS experience. Exposing QoS requirements in Web applications effectively guides the underlying Web runtime to determine how to deliver the target QoS experience while minimizing the energy consumption. `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

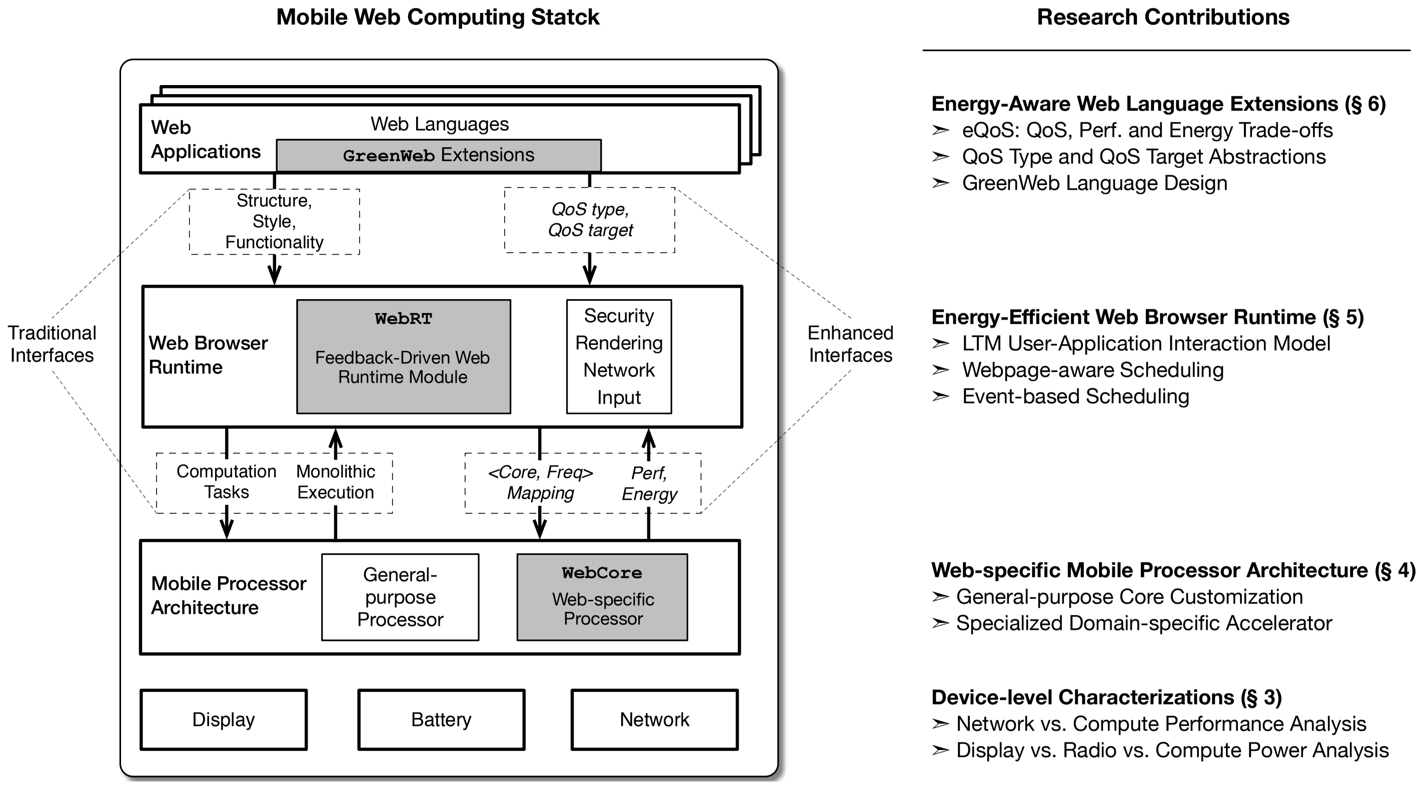

Mobile Web Computing Statck
Web Applications
Web Languages
GreenWeb Extensions
Structure, Style, Functionality
QoS type, QoS target
Traditional Interfaces
Enhanced Interfaces
Web Browser Runtime
WebRT
Feedback-Driven Web Runtime Module
Security
Rendering
Network
Input
Computation Tasks
Monolithic Execution
<Core, Freq> Mapping
Perf, Energy
Mobile Processor Architecture
General-purpose Processor
WebCore
Web-specific Processor
Display
Battery
Network
Research Contributions
Energy-Aware Web Language Extensions (§ 6)
➢ eQoS: QoS, Perf. and Energy Trade-offs
➢ QoS Type and QoS Target Abstractions
➢ GreenWeb Language Design
Energy-Efficient Web Browser Runtime (§ 5)
➢ LTM User-Application Interaction Model
➢ Webpage-aware Scheduling
➢ Event-based Scheduling
Web-specific Mobile Processor Architecture (§ 4)
➢ General-purpose Core Customization
➢ Specialized Domain-specific Accelerator
Device-level Characterizations (§ 3)
➢ Network vs. Compute Performance Analysis
➢ Display vs. Radio vs. Compute Power Analysis

Fig. 1.1: Overview of my cross-layer research contributions.

specified user QoS requirements. Although WebRT is a generic runtime design, I demonstrate a prototype implementation based on the asymmetric chip-multiprocessor (ACMP) hardware architecture. ACMP exposes two new architecture-level abstractions: core type and core frequency. WebRT leverages the new abstractions and dynamically provisions the hardware resources according to user QoS requirements for energy savings. In addition, WebRT also continuously monitors the runtime execution behaviors to enable feedback-driven optimizations, which is critical considering the interactive nature of mobile applications.

- **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 and data communication patterns. A key design goal of WebCore is maintain general-purpose programmability, which is vital to ensure its applicability to the complex Web software stack. Overall, WebCore deepens the heterogeneity of the mobile processor architecture and enlarges the performance-energy trade-off space that the Web runtime can take advantage of.

1.2 Long-term Impact

Mobile hardware and Web software ecosystems undergo rapid design cycles to keep up with constant innovations. It is vital to ensure that any research contributions to this domain have long-term impact, or they perish.

The long-term impact of my work lies in two fundamental aspects. First, the problem that I study is a long-term research agenda. The key challenge that my research focuses on, i.e., performance and energy-efficiency, will always be at the forefront of mobile computing research. As the battleground of mobile computing gradually shifts into even smaller form factors such as wearables and Internet-of-Things (IoT) devices, improving performance and energy-efficiency of mobile computing is ever important.

Second, my proposed techniques have long-term applicability because they focus on the fundamental computation layers of Web technologies rather than being tied to the specifics of today's systems. For instance, `WebCore` proposes a hardware units that optimizes CSS processing, which is a cornerstone technology that remains largely unchanged as new Web standards and specifications come and go. Similarly, the designs of `WebRT` and `GreenWeb` are also generally applicable because they are do not rely on a particular form of the underlying processor (micro-)architecture or application features.

1.3 Research Scope

The scope of mobile Web computing is broad and becoming increasingly rich. It involves two critical components: compute and network. The compute component can be further classified by approaches that are either client-centric or based on cloud-offloading. Figure 1.2 shows the hierarchy of the mobile Web scope. My research judiciously focuses on the client-side compute. In other

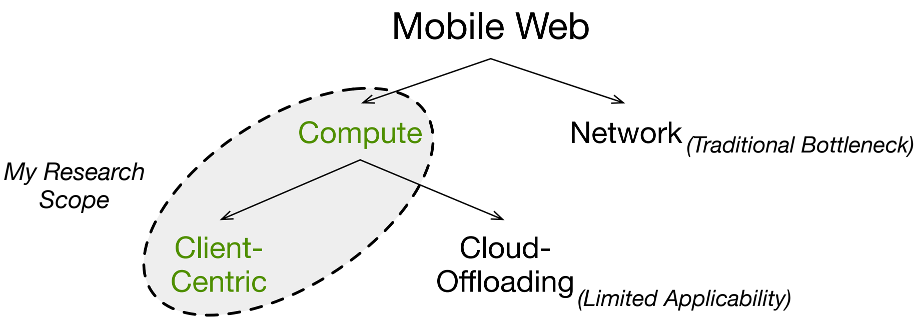

Mobile Web
Compute
Network (Traditional Bottleneck)
My Research Scope
Client-Centric
Cloud-Offloading (Limited Applicability)

Fig. 1.2: The mobile Web computing scope. My research takes a compute-driven, client-centric approach.

words, it takes a compute-driven and client-centric approach. This section discusses my rational. The goal here is not to dismiss research in network and cloud computing community, but to explain the trade-offs between different approaches and thereby set the context for my work.

Compute-versus-Network Traditionally when the scope of Web computing was merely about serving static webpages, Web performance was predominantly bottlenecked by network capability because very little processing was involved. However, this trend is changing. Over the past decade cellular network technology has improved dramatically. For example, the round-trip time is improved by two orders of magnitude from 2G to LTE [111]. Meanwhile, the computational requirements posed by new Web technologies (e.g., CSS3, HTML5, WebGL) keep increasing. For instance, under the same network condition the processing time for loading the same website from different years over the past decade has increased by as much as 10X [185]. The combined effect of faster network performance and higher computation demand implies that future mobile Web performance will be unattainable without improving the compute capability. An in-depth computer-versus-network bottleneck analysis can be found in Chapter 3.1.

Client-versus-Cloud Compute in mobile Web has been primarily carried out by client-side devices only. Recently researcher have started investigating a new compute paradigm where part of the computation is offloaded to cloud platforms through wireless connections—the so called mobile cloud computing (MCC). Although MCC is a promising approach that extends the capability of mobile devices for computation-intensive applications, it has three major limitations. First, today's Web applications are extremely dynamic where both data and code can be generated at runtime depending on user-specific information (e.g., via sensors). The dynamic nature of Web applications leads to frequent synchronizations between client and cloud that potentially undermine the performance and energy-efficiency benefits that MCC brings. Second, MCC assumes the availability of wireless connections, which limits its usage scenario. Third, MCC raises security and privacy concerns as data and computation are transmitted over the network.

The limitations stated above indicate that a handful questions need to be addressed for MCC to succeed. That said, future mobile Web computing systems will most certainly incorporate certain aspects of cloud-offloading, a quantitative trade-off study of which is warranted but beyond my scope.

1.4 Dissertation Organization

The rest of my dissertation is organized as follows. Chapter 2 introduces the preliminary knowledge of Web computing. Chapter 3 quantitatively demonstrates the need for high-performance and energy-efficient computation in the mobile Web, which directly motivates the research theme of my work. Chapter 4, Chapter 5, and Chapter 6 describe the proposed `WebCore`, `WebRT`, and `GreenWeb` at the architecture, runtime, and programming language layer, respectively. Chapter 7 provides a retrospective and prospective view of my dissertation work. The retrospective part summarizes the principles distilled from this work on building a high-performance while energy-efficient mobile Web computing system; the prospective part suggests next steps for generalizing the principles and outlines potential research items for future work.

1.5 Previously Published Material

This dissertation contains materials that are previously published in peer-reviewed conferences and journals:

Chapter 3. The network-versus-computer analysis in Chapter 3.1 contains results from the following paper: *The Role of the CPU in Energy-Efficient Mobile Web Browsing.* Yuhao Zhu, Matthew Halpern and Vijay Janapa Reddi. In IEEE Micro, Jan/Feb 2015, 35(1):26-33 [184]. The power and energy characterizations in Chapter 3.2 contains results from the following paper: *Mobile CPU's Rise to Power: Quantifying the Impact of Generational*

Mobile CPU Design Trends on Performance, Energy, and User Satisfaction. Matthew Halpern, Yuhao Zhu and Vijay Janapa Reddi. In High Performance Computer Architecture (HPCA), 2016 [104].

Chapter 4. The design and implementation of `WebCore` are based on the following paper: *WebCore: Architectural Support for Mobile Web Browsing.* Yuhao Zhu and Vijay Janapa Reddi. In International Symposium on Computer Architecture (ISCA), 2014 [186]. Chapter 4 also contains results from the following journal paper: *Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing.* Yuhao Zhu and Vijay Janapa Reddi. In ACM Transactions on Computer Systems (TOCS), March 2017, 35(1):1 [188].

Chapter 5. The fundamental idea of `WebRT` is based on the following position paper: *Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing.* Yuhao Zhu, Aditya Srikanth, Jingwen Leng and Vijay Janapa Reddi. In Computer Architecture Letters (CAL), Oct 2012, 13(1):33-36 [189]. The webpage-aware scheduler described in Chapter 5.4 draws upon *High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems.* Yuhao Zhu and Vijay Janapa Reddi. In High Performance Computer Architecture (HPCA), 2013 [185]. The event-based scheduler in Chapter 5.5 draws upon *Event-based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications.* Yuhao Zhu, Matthew Halpern and Vijay Janapa Reddi. In High Performance Computer Architecture (HPCA), 2015 [183].

Chapter 6. The `GreenWeb` language extensions and the AutoGreen annotation framework are based on the following paper: *GreenWeb: Language Extensions for QoS-aware Energy-Efficient Mobile Web Computing.* Yuhao Zhu and Vijay Janapa Reddi. In Programming Language Design and Implementation (PLDI), 2016 [187].

Chapter 2

Web: A Universal Application Platform

In this section, I first present the broad scope of the Web computing that this dissertation focuses on (Chapter 2.1). I then briefly introduce Web languages and the Web browser runtime (Chapter 2.2). Overall, I show that the Web has become a cornerstone technology in today's mobile computing era. Its evolution is largely driven by innovations made in programming languages and system design. These observations motivate my effort on designing a holistic energy-efficient mobile Web computing substrate.

2.1 The Scope of the Web

Web applications are applications developed using Web languages, including HTML, CSS, and JavaScript. Originally, webpages running in a Web browser were the only form of Web application. The scope of the Web today has been greatly expanded beyond webpages to a universal application development platform. The driving force is Web's "write-once, run-anywhere" feature that tackles the notorious device fragmentation issue [161]. Strategy Analytics reported that by the year 2015 63% of all business mobile applications are based on Web technologies [162].

Mobile system vendors are actively embracing Web technologies. Both iOS and Android provide developers APIs that expose Web browser functionalities [4, 22]. This allows developing “hybrid” applications that are internally based on Web technologies, but are wrapped by a native shell. Such a development strategy has been widely adopted by popular mobile Apps such as Uber and Instagram [52]. In this dissertation, the scope of Web application extends beyond webpages to also include such hybrid applications.

2.2 Web Languages and The Web Browser Runtime

HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript are the three fundamental languages for Web development. In a nutshell, HTML describes the structural information of a Web application by building a Document Object Model (DOM) tree [17], in which each node represents a Web application element. CSS describes an application’s style information by declaring visual properties of each DOM tree node. JavaScript specifies an application’s dynamic behavior by defining callback functions to execute when certain user interactions are triggered on DOM nodes.

To enable portability of Web applications, the Web browser acts as a “virtual machine” or a runtime system layer that dynamically translates HTML, CSS, and JavaScript to different platforms. Figure 2.1 shows the overall flow of execution within any typical Web browser, which typically consists of two core modules: a rendering engine (e.g., WebKit for Chrome and Gecko for Firefox) that translates HTML and CSS, and a JavaScript engine

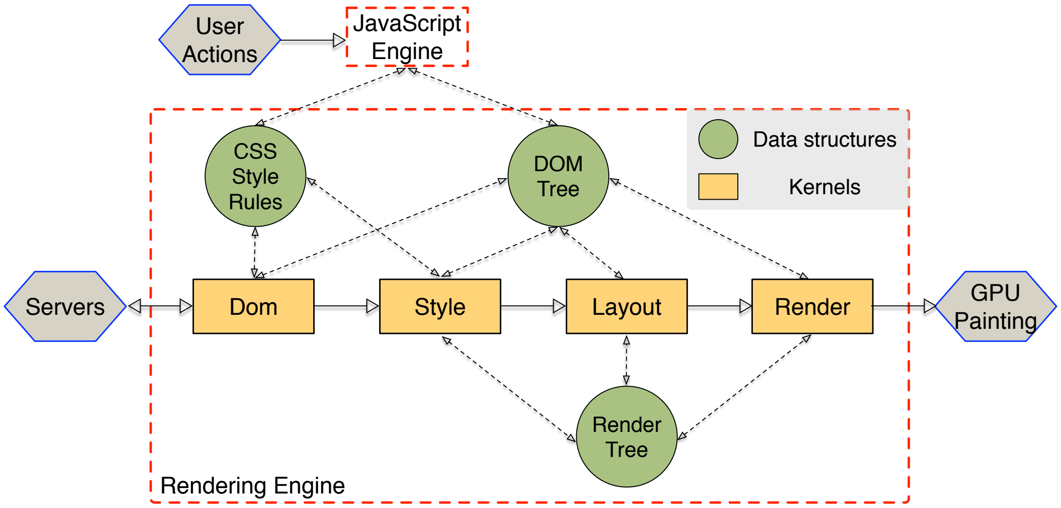


Fig. 2.1: A typical Web browser architecture.

that executes JavaScript code.

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

The *Dom* kernel is in charge of parsing the webpage contents. Specifically, it constructs the DOM tree from the HTML files, and extracts the CSS style rules from the CSS files. Given the DOM tree and CSS style rules, the *Style* kernel computes the webpage's style information and stores the results in the render tree. Each render tree node corresponds to a visible element in the webpage. Once the style information of each webpage element is calculated, the *Layout* kernel recursively traverses the render tree to decide each visible element's position information based on each element's size and relative positioning. The final $< x, y >$ coordinates are stored back into the render tree. Eventually, the *Render* kernel examines the render tree to decide the z-ordering of each visible element so that they can be displayed in the correct overlapping order.

Over the past two decades, language evolution and Web runtime design have been constantly driving Web innovations [36, 42]. As a result, current Web standards such as HTML5 and CSS3 enable ever-richer functionalities, such as offline storage, media playback, and geolocation, that are the core in today's mobile applications. Web language and browser design innovations will continue to be the key enabler for next-generation Web computing.

Chapter 3

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 work 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 [49]. 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 fact, one of my dissertation contributions is the accelerator design of a key Web computation kernel based on its CPU execution behaviors.

In the rest of this chapter, I first show that the overall mobile Web performance depends increasingly on the computational capability of mobile CPUs, indicating the need for a high-performance computation (Chapter 3.1). I then show that mobile devices' power consumption is increasingly dominated by CPUs, calling for an energy-efficient computation (Chapter 3.2).

3.1 The Need for Compute Performance 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 Network performance is typically evaluated in two metrics: latency and bandwidth. Prior work has shown that in the mobile context, network latency—typically evaluated by round-trip time (RTT)—has a much more significant impact than network bandwidth [97, 172]. Therefore, we focus only on the latency aspect of network performance.

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

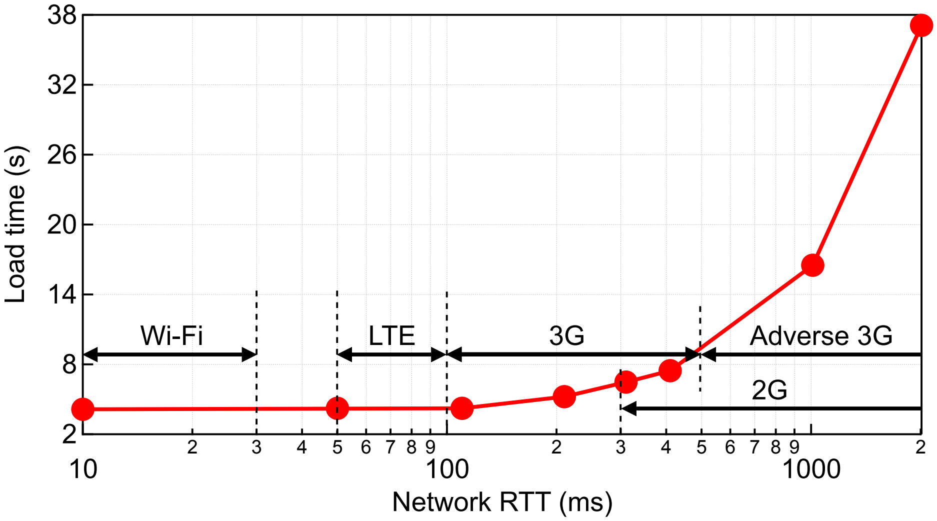


Fig. 3.1: Webpage load time with respect to changing network latency. Each marker corresponds to an RTT value. We also superimpose the round-trip time (RTT) range for different cellular technologies derived from both technical specifications as well as real measurements in the field [1, 97].

inject delay. We then use Wi-Fi on the smartphone to access the webpages. The delay injection lets us mimic a wide range of network latencies because Wi-Fi has significantly lower latency than the current 4G/LTE network. This methodology is well-established to control cellular network latency [172].

Holding the CPU performance at its peak, Figure 3.1 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 [1, 97]. We observe that reducing the network latency initially 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. However, 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* for CPU performance, it is *not* our intention to study the impact of a particular CPU's frequency itself. Figure 3.2 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

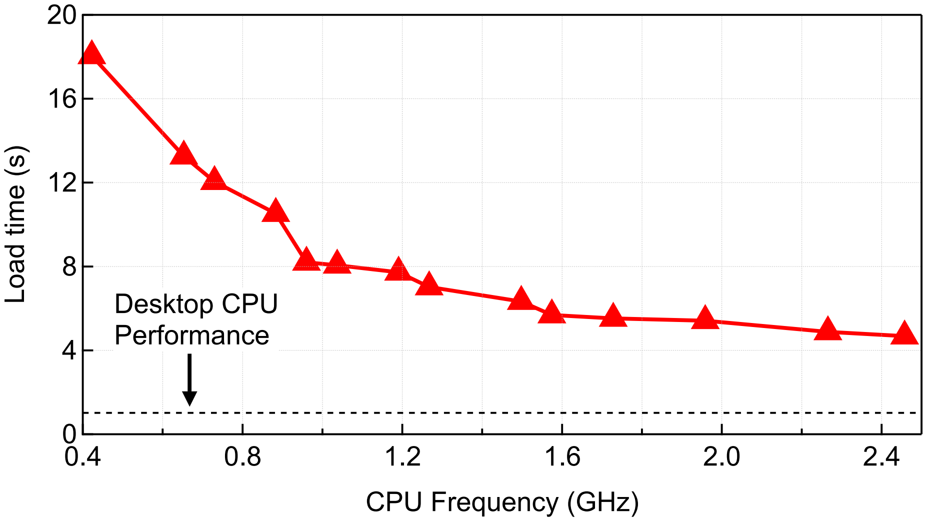


Fig. 3.2: Webpage load time with respect to CPU frequency on a Galaxy S5 smartphone. The markers represent CPU frequencies, which range from 0.4 GHz (left) to 2.5 GHz (right). We also overlay the load time on a desktop CPU (the dotted line) for comparison purpose.

pipeline is already saturated, and the memory and interconnection become the microarchitecture-level bottlenecks [54]. 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 Figure 3.2. 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 [25]. Note, however, that we do *not* claim that network latency is irrelevant. When the network deviates from an ideal low-latency condition, or in emerging markets where high-latency network accesses are prevalent [123], mobile Web performance is indeed constrained by the network latency.

3.2 The Need for Energy-Efficient Computation

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 [104] that I collaborated on.

Energy Constraint Battery technology has not experienced Moore's law-like improvements because of fundamental physics limitations [158]. As a result, the density of lithium-ion batteries has improved by only about 10% per year [62]. Thus, the battery capacity of today's mobile devices is deter-

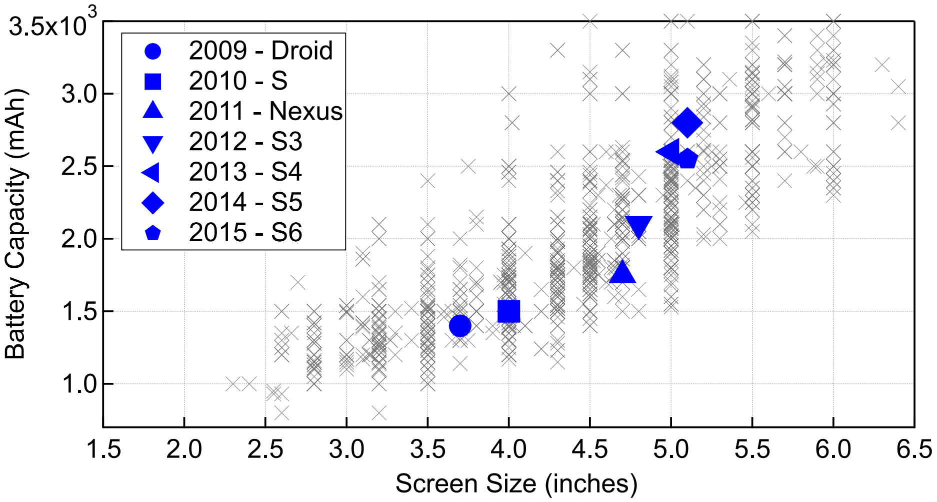


Fig. 3.3: The correlation between smartphone battery capacities and their screen sizes. There is an almost linear relationship over the time.

mined by the battery's volume, which is largely dictated by the device's screen size [6]. Using the smartphone as an example of a start-of-the-art mobile device, Figure 3.3 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 [47], 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. Figure 3.4 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,

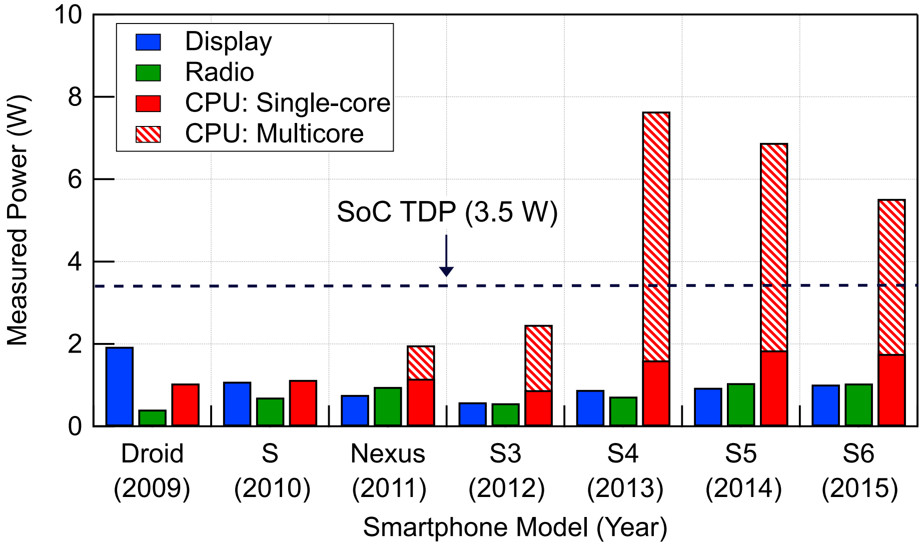


Fig. 3.4: CPU measured peak power consumption has increased significantly compared to other key mobile device components over the years. In particular, the multicore CPU power alone can exceed SoC-level TDP.

the seven phones represent how cutting-edge smartphone technologies have progressed over time. The results are collected while running a standard Web benchmark, Sunspider [34], for the CPU(s), and dedicated benchmarks for the other components [28, 73]. We used the Monsoon power monitor to measure the seven smartphones' power consumptions at the battery level.

We make two important observations from Figure 3.4. First of all, CPU has become a major power consumer in 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 power consumption exceeds the entire mobile device's thermal design power (TDP) even without considering the power consumption of radio, display, and the rest of the SoC.

Second, while other mobile device components are becoming more power-efficient over time due to their respective technological advancements [78], the CPU's power has risen excessively. The continuous mobile CPU power increase is a direct result of the mobile CPU design strategy, namely simply adopting desktop-like design techniques, such as aggressive single-core microarchitecture enhancement and multicore scaling, to improve performance at the expense of high power consumption. A recently study shows that mobile CPUs incorporated over 20 years of desktop CPU design techniques in just about seven years [104]. The inevitable consequence of such a design methodology is that as the Dennard Scaling [84] comes to its end, the performance improvement can no longer make up for the additional power consumption [104], eventually making the mobile CPU extremely energy-inefficient.

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.

Chapter 4

WebCore: A Mobile Processor Architecture Substrate for Web Computing

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 [105, 131, 176]. 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 [145]. 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 [105], convolution [152]).

Following the architecture design philosophy of balancing general-purpose programmability and domain-specific specialization, we propose `WebCore`, *a general-purpose core customized and specialized for mobile Web computing.* In comparison to prior work that either takes a fully software approach on general-purpose processors [75, 137] or a fully hardware specialization approach [63], our design strikes a balance between the two. On one hand, `WebCore` retains the flexibility and programmability of a general-purpose core. It naturally fits in the multicore SoC that is already common in today's mainstream mobile devices. On the other hand, it achieves energy-efficiency improvement via modest hardware specializations that create closely coupled datapath and data storage.

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

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

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

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

The rest of this chapter is organized as follows. We first describe our experimental setup including software/hardware infrastructure and application selection in Chapter 4.1. We then describe the design-space explorations that allow us to identify sources of inefficiency in existing general-purpose processors and customize them for mobile Web applications in Chapter 4.2. Building on top of the customized general-purpose designs, we further propose the two new specialization techniques in Chapter 4.3 and Chapter 4.4. We show that our proposed `WebCore` achieves significant performance and energy-efficiency improvement over existing designs in Chapter 4.5. We review related work in Chapter 4.6.

4.1 Experimental Setup

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

Web Browser We focus on the popular WebKit [173] rendering engine used in Google Chromium (Version 30.0) for our studies. WebKit is also widely used by other popular mobile browsers, such as Apple's Safari and Opera.

Benchmarked Web Applications We pay close attention to the choice of webpages to ensure that the WebCore design is not misled. We mine through the top 10,000 websites as ranked by Alexa [55] and pick the 12 most representative websites. All except one happen to rank among Alexa's

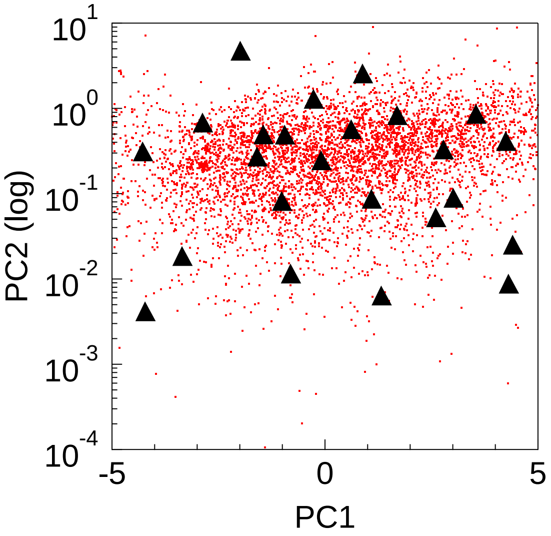


Fig. 4.1: We pick 24 representative webpages from 10,000 of the hottest webpages as per `www.alexa.com`

top 25 websites. The 12 benchmarked websites also cover 10 of BBench's 11 webpages [101]. Chapter 4.5 lists the website names. We refer interested readers to Chapter 4.6.3 for a discussion of BBench.

We consider not only the mobile version of the 12 websites, but also their desktop counterparts. Many mobile users still prefer desktop-version websites for their richer content and experience [65, 163]. Moreover, many mobile devices, especially tablets, typically load the desktop version of webpages by default. As webpage sizes exceed 1 MB [64], we must study mobile processor architectures that can process more complex content and not just simple mobile webpages.

We study 24 distinct webpages in total. The 24 benchmarked webpages are representative because they capture the webpage variations in both webpage-inherent and microarchitecture-dependent features. To prove this, we performed principal component analysis (PCA), which is a statistical method that reduces the number of inputs without losing generality [86]. PCA transforms the original inputs into a set of principal components (PC) that are linear combinations of the inputs. In our study, PCA calculates four PCs from about 400 distinct features. These four PCs account for 70% of the variance across all of the original 10,000 webpages. Figure 4.1 shows the results for two major components, PC1 and PC2. IPC (microarchitecture-dependent feature) is the single most significant metric in PC1, and the number of DOM tree nodes (webpage-inherent feature) is the most significant metric in PC2. The triangular dots represent our webpages. They cover a very large spread of the top 10,000 webpages in the Internet.

Performance Metric We focus on the initial loading of Web applications. This is because user QoS experience is strongly tied to the initial load time in Web applications. For instance, it is estimated that 79% of online shoppers will not return to the website with slow load time [122].

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

Simulators We assume the x86 instruction set architecture (ISA) for our study. Prior work shows that the ISA does not significantly impact energy efficiency for mobile workloads [66]. Therefore, we believe that our microarchitecture explorations are generally valid across ISAs. We use Marss86 [149], a cycle-accurate simulator, in full-system mode to faithfully model all the network and OS activity. Performance counters from Marss86 are fed into McPAT [127] for power estimation.

4.2 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 (Chapter 4.2.1). 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 (Chapter 4.2.2). 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 (Chapter 4.2.3).

4.2.1 Design Space Exploration

Design Space Specification We define the set of tunable microarchitectural parameters in Table 4.1. 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 [70]. We also do not consider single-issue out-of-order processors, which are known to be energy inefficient [58]. In total, we consider over 3 billion design points.

We intentionally relax the design parameters beyond the current mobile systems in order to allow an exhaustive design space exploration. For example, we consider up to 128 KB L1 cache design whereas most L1 caches in existing mobile processors are 32 KB in size. Also, since thermal design power (TDP) is important for mobile SoCs, we eliminate overly aggressive designs with more than 2 W TDP.

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

We use a constant memory latency to model the memory subsystem because we do not observe significant impact of the memory system on the mobile Web browsing workload. According to hardware measurements on the Cortex-A15 processor using ARM's performance monitoring tool Streamline [57], the MPKI for the L2 cache across all the webpages is below 5. We observe similar

Table 4.1: Microarchitecture design-space parameters. The first column shows the parameters that are considered in our DSE. The second column shows the metric that the value of each parameter is measured. The i::j::k in the third column denotes values ranging from i to k at steps of j

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

low L2 MPKI, i.e. low main memory pressure, in our simulations. Therefore, we use a simpler memory system to further trim the search space.

Statistical Inference Method It is not feasible to simulate billions of the design points that we consider simply due to time constraints. Therefore, we leverage the statistical inference technique that trains predictive models using a small number of samples. Such models reflect how different microarchitecture parameters, both individually and collectively, influence performance and power consumption. Statistical inference methods have been used successfully in the past for architecture design-space exploration [100, 125].

In particular, we use linear regression modeling [108] to construct our predictive models. A linear regression model can be formulated as in Equation. 4.1, where y denotes the response, $x = x_1, ..., x_p$ denote p predictors, and $\beta = \beta_0, ..., \beta_p$ denote corresponding coefficients of each predictor. The *least squares method* is used to solve the regression model by identifying the best-fitting β that minimizes the residual sum of squares (RSS) [109]. In our case, the response y is either performance (measured in terms of instruction per cycle, IPC) or power, and the predictors x_i are microarchitecture structures listed in Table 4.1.

$$y = \beta_0 + \sum_{i=1}^{p} x_i \beta_i \tag{4.1}$$

We find that 2,000 *uniformly at random* (UAR) samples of microarchitecture configurations from the design space are sufficient in our case to

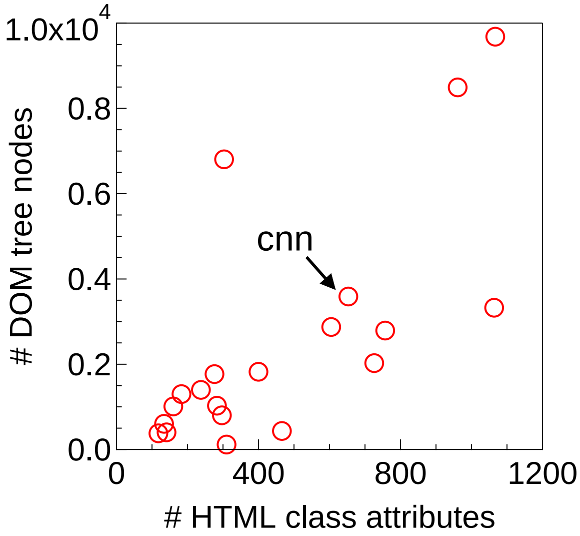


Fig. 4.2: `www.cnn.com` is a representative webpage from our benchmark suite because it is almost the centroid.

construct robust models. We also obtain 500 additional UAR samples from the cache design space (both L1 and L2) to reinforce the credibility of instruction and data cache design predictions. We perform cross-validation of the model (i.e., we partition a sample dataset into complementary subsets, and perform analysis on one subset and validate the analysis on the other subset), and then obtain additional samples from the design space for full evaluation.

In order to derive general conclusions about the design space and optimize for the common case, in this section we present only our in-depth analysis for the representative website `www.cnn.com`. Figure 4.2 compares `www.cnn.com` with other webpages to demonstrate that it is indeed representative of the other benchmarked webpages. The x-axis and y-axis represent the number of DOM tree nodes and the number of class attributes in HTML. These are

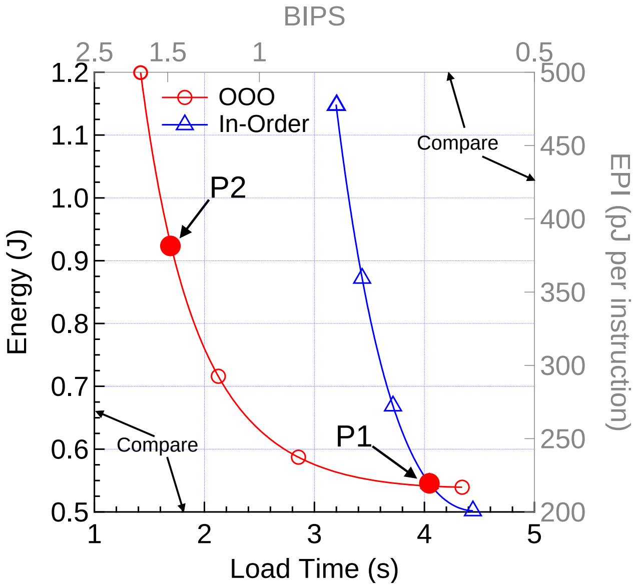


Fig. 4.3: In-order versus out-of-order Pareto optimal frontiers.

the two webpage characteristics that are most correlated with a webpage's load time and energy consumption [185]. As the figure shows, `www.cnn.com` is roughly the centroid of the benchmarked webpages, and thus we use it as a representative webpage for the common case.

We construct predictive models for out-of-order and in-order design space separately because microarchitecture structures have different impact on performance and power in in-order and out-of-order pipelines. In general, the out-of-order models' error rates are below 6.0%. The in-order models (not shown) are more accurate because of their simpler design. On average, the in-order performance and power models' errors are within 5% and 2%, respectively.

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

Design space exploration helps customization at the "macro-architecture" level, i.e., determining between in-order and out-of-order designs. We understand the difference between in-order and out-of-order design space by examining their Pareto optimal frontiers. Design points on a Pareto optimal frontier reflect different optimal design decisions given specific performance/energy targets. The Pareto-optimal is more general than the (sometimes overly specific) EDP, ED^2P metrics, etc. Design configurations optimized for such metrics have been known to correspond to different points on the Pareto-optimal frontier [58]. Figure 4.3 shows the Pareto-optimal frontiers of both in-order and out-of-order designs between energy and performance. We use energy per instruction (EPI) for the energy metric, and million instructions per second (MIPS) as the performance metric.

We make two important observations from Figure 4.3. First, the out-of-order design space offers a much larger performance range ($\sim$1 BIPS between markers P1 and P2, see top x-axis) than the in-order design space ($<$ 0.5 BIPS), which reflects the out-of-order's flexibility in design decisions. Second, the out-of-order design frontier is flatter around the 4-second webpage load time range (see marker P1) than in the in-order design, which indicates that the out-of-order design has a much lower marginal energy cost. The observation indicates that processor architects can make design decisions based on the different performance goals without too much concern about the energy budget. In contrast, the in-order design space quickly enters the region

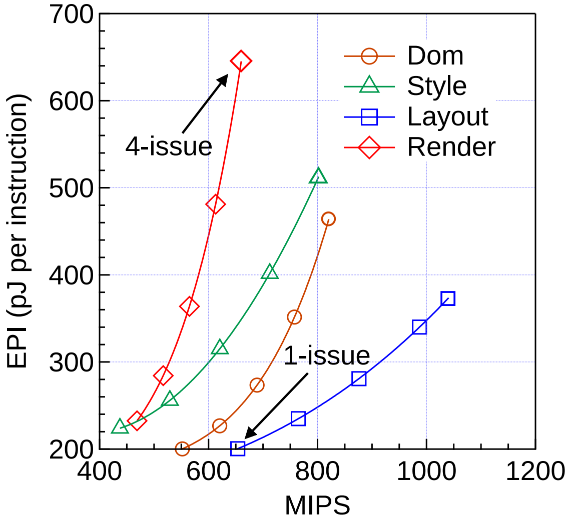


Fig. 4.4: In-order Pareto optimal frontier for each kernel.

of diminishing returns (i.e., sharp increase in energy consumption) as we push toward webpage load times that are less than 4 seconds. In other words, the in-order design has a low marginal performance value (or equivalently high marginal cost of energy).

To understand the 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. Figure 4.4 and Figure 4.5 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

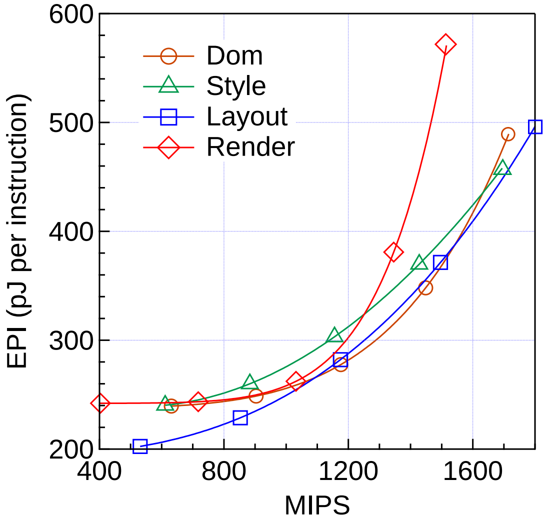


Fig. 4.5: Out-of-order Pareto optimal frontier for each kernel.

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 [58]. 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.

4.2.3 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 Figure 4.3. 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 4.2 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 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 [101].

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.

Table 4.2: Microarchitecture configurations for P1 and P2 in Figure 4.3. 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 [53] and ARM's public presentation [56].

	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

4.3 Style Resolution Unit

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

The SRU is an accelerator for the critical *Style* kernel within the Web browser rendering engine. The SRU design is based on the observation that the *Style* kernel has abundant fine-grained parallelism that is hidden in a software implementation but can be captured by a dedicated hardware structure (Chapter 4.3.1). To exploit the inherent fine-grained parallelism, the SRU employs a multi-lane parallel architecture, which greatly reduces the instruction delivery overhead. To reduce the data feeding pressure, the SRU is tightly coupled with a small scratchpad memory that brings operands closer to the SRU (Chapter 4.3.2). To maintain general-purpose programmability, these new hardware structures are accessed via a set of high-level language APIs. The APIs are implemented through a runtime library with only slight modification to the current browser implementation (Chapter 4.3.3).

4.3.1 Motivation

Optimizing the *Style* kernel would improve the overall energy efficiency the most for the following reasons. The *Style* kernel is the most time-consuming task in the rendering engine. In our profiling, it consumes 35% of the total rendering engine execution time. it also dominates the energy consumption by consuming 40% of the total energy.

In order to mitigate the instruction delivery and data communication overhead of the *Style* kernel, we propose a special functional unit called the *Style Resolution Unit* (SRU) that is tightly coupled with a small scratchpad memory. The SRU exploits fine-grained parallelism to reduce the amount of instructions and potential divergences. The scratchpad memory reduces data communication pressure by bringing operands closer to the SRU.

The *Style* kernel consists of two phases: a matching phase and an applying phase. Figure 4.6 shows the pseudo-code of the two phases. Previous work [75, 137] focuses on parallelizing the matching phase. However, in our profiling, we find that the applying phase takes nearly twice as long to execute as the matching phase. Therefore, we focus on the applying phase. The applying phase takes in a set of CSS rules (`matchedRules`) as input, iterates over each rule in the correct cascading order [171] to calculate each style property's final value (e.g., the exact-color RGB values, font width pixels). The final values are stored back to the Render tree (the `RenderStyle` array).

The key observation we make in the applying phase is that there are two

```
// matching phase
matchedRules = matching(DOMTree, DOMNodeId, CSSRules);

// applying phase
foreach (rule in matchedRules) {
  foreach (property in rule) {
    switch (property.ID) {
      case Font:
        RenderStyle[Font] = FontHandler(property, DOMNodeId);
        break;
      case Color:
        RenderStyle[Color] = ColorHandler(property, DOMNodeId);
        break;
      …
      case N: ...
    }
  }
}
```

Fig. 4.6: Pseudo-code of the *Style* kernel. It consists of a matching phase and an applying phase. SRU accelerates the applying phase, which takes about two-thirds of the *Style* kernel execution time.

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

RLP comes from the following. In order to maintain the correct cascading order, each rule contained in the input data structure must be sequentially iterated from the lowest priority to the highest, so that the higher-priority rules can override the lower-priority rules. However, in reality, we could speculatively apply the rules with different priorities in parallel, and select the one

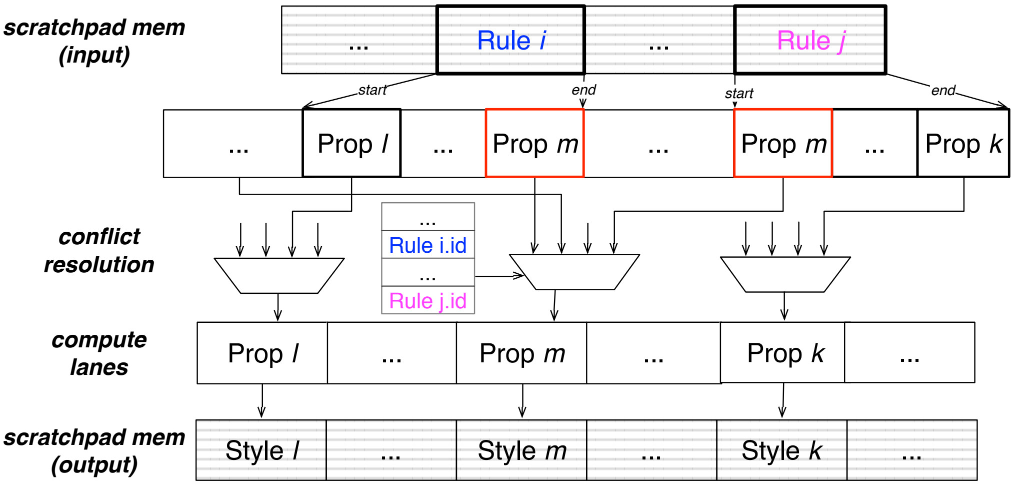


Fig. 4.7: SRU coupled with scratchpad memories.

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

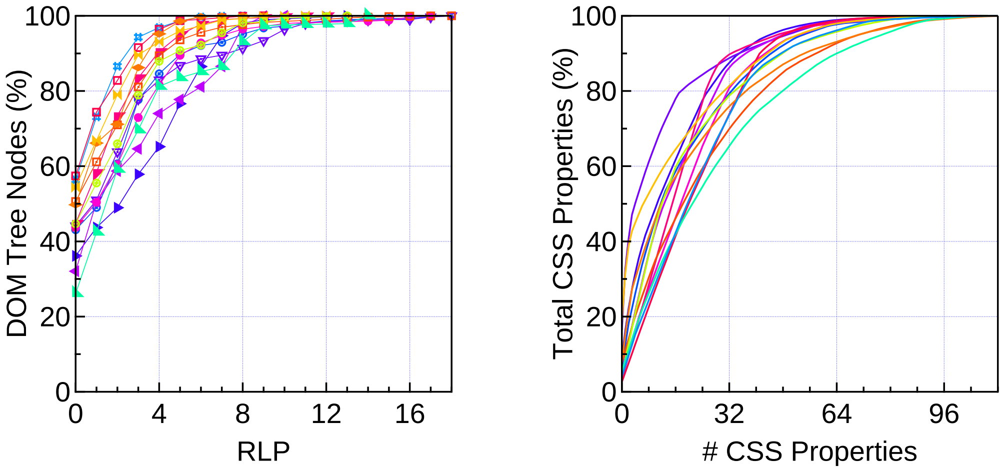
4.3.2 Hardware Design

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

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

Figure 4.7 shows the structure of the SRU with scratchpad memory for input and output data. SRU has multiple lanes, with each lane dealing with one CSS property. Assume Rule i and Rule j are two rules from the input that are residing in the scratchpad memory. Rule i has higher priority than Rule j. Prop l and Prop m are two properties in Rule i. Similarly, Rule j has properties Prop k and Prop m. Prop l and Prop k can be executed in parallel using different SRU lanes because they do not conflict with each other. However, Prop m is present in both rules, and as such it causes an SRU lane conflict, in which case the MUX selects the property from the rule with the highest priority, which in our example is Rule i.

Design Considerations A hardware implementation can have only a fixed amount of resources. Therefore, the number of SRU lanes and the size of the scratchpad memory is limited. Prior work [185] shows that the number of matched CSS rules and the number of properties in a rule can vary from one webpage to another. As such, a fixed design may overfeed or underfeed the SRU if the resources are not allocated properly.



(a) RLP analysis.

(b) CSS property analysis.

Fig. 4.8: Analysis of RLP and CSS properties across webpages.

We profile the webpages to determine the appropriate amount of resource allocation required for the SRU. Profiling indicates that 90% of the time, the RLP is below or equal to 4 (Figure 4.8a). 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 (Figure 4.8b). Thus, we implement a 32-wide SRU where each lane handles one hot CSS property. Due to these considerations, the input and output scratchpad memories are each 1 KB in size.

Furthermore, not all of the properties are delegated to the SRU. For example, some style properties require information on the parent and sibling nodes. To avoid complex hardware design for recursions and loops with unknown iterations, we do not implement them in our SRU prototype. The runtime library performs these checks, which we discuss later in Chapter 4.3.3. Despite the trade-offs we make, about 72.4% of the style rules across all the benchmarked webpages can utilize the SRU.

4.3.3 Software Support and Programmability

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

```
// matching phase
matchedRules = matching(DOMTree, DOMNodeId, CSSRules);

// applying phase
Style_Apply(DOMNodeId, matchedRules);
```

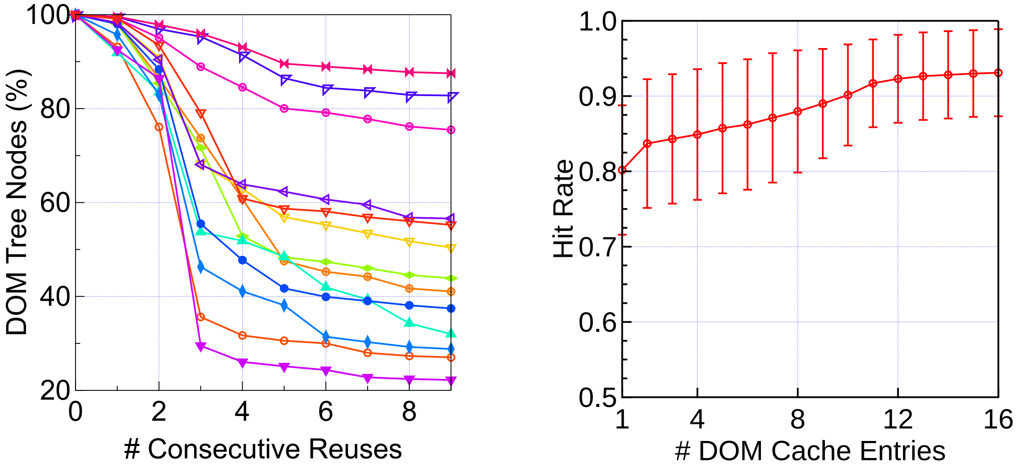
Fig. 4.9: Pseudo-code of the *Style* kernel with the new API.

Programmers trigger the style resolution task by issuing a `Style_Apply(Id, Rules)` API, in which `Id` represents a DOM tree node ID and `Rules` represents matched CSS rules produced by the matching phase. Figure 4.9 illustrates the pseudo-code of the *Style* kernel using the provided API. Comparing against the original code in Figure 4.6, we notice that the matching phase is not changed while the applying phase is greatly simplified with the `Style_Apply` API.

One key task of this API implementation is to examine all the CSS properties of a particular DOM node because not all the CSS properties are implemented in the SRU (as discussed in Chapter 4.3.2). 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.

4.4 Browser Engine Cache

To further improve the energy-efficiency of date feeding, we propose the browser engine cache. It is based on the observation that Web applications' accesses to principal data structures, such as the DOM tree and the Render tree, exhibit heavy data reuse and predictable access pattern (Chapter 4.4.1). Based on such an observation, the browser engine cache uses a small hardware memory structure coupled with a lightweight software-based cache management layer to provide energy-efficient data access (Chapter 4.4.2). In addition, similar to SRU, we also provide a set of high-level language APIs that allow Web browser developers to easily access the browser engine cache (Chapter 4.4.3).



(a) DOM node reuse behavior.

(b) DOM node access hit rate.

Fig. 4.10: DOM tree access behavior across webpages.

4.4.1 Motivation

The DOM tree and Render tree are the two most important data structures because they are shared across different kernels. We propose the Browser Engine Cache to improve the energy-efficiency of accessing them. Specifically, the browser engine cache consists of a DOM cache and a Render cache for the DOM tree and Render tree, respectively. We use the DOM to explain our locality observation. Similar analysis and design principles also apply to the render cache. Note that the browser engine cache focuses on improving the energy efficiency of data feeding. We will discuss techniques for improving the performance aspect of data accesses in Chapter 4.6.

The energy inefficiency of the traditional cache is best embodied in the performance-oriented design P2 in Table 4.2. P2 requires a larger data cache (64 KB) compared to a traditional mobile core. Although a large cache achieves a high hit rate of 93%, it leads to almost one-fourth of the total energy consumption. However, through careful characterizations, we find that accesses to the DOM/Render tree have strong locality and regular access pattern such that they can benefit from a small and energy-efficient cache memory, rather than the large power-hungry traditional caches. Let us explain our observations below.

First, we find that data accesses to the DOM tree have heavy reuses. Figure 4.10a shows the cumulative distribution of DOM tree node reuse. Each (x, y) point corresponds to a portion of DOM tree nodes (y) that are consecutively reused at least a certain number of times (x). About 90% of the DOM tree nodes are consecutively reused at least three times, which reflects strong data locality. This indicates that a very small cache can achieve the similar hit rate as a regular cache, but with much lower power.

Second, we find that the accesses to the DOM tree have regular stream-like patterns. To illustrate this, Figure 4.11 shows two representative data access patterns to the DOM tree from `www.sina.com` and `www.slashdot.org`. Each (x, y) point is read as follows. The x-th access to the DOM tree operated on the y-th DOM node. We observe a common streaming pattern. Such a streaming pattern is due to the intensive DOM tree traversal that is required by many rendering engine kernels. For example, in order to match CSS rules with descendant selectors such as "`div p`," which selects any <`p`> element that is a descendant of <`div`> in the DOM tree, the *Style* kernel must traverse the

DOM tree, one node at a time, to identify the inheritance relation between two nodes. Similarly, the *Layout* kernel must traverse the Render tree (recursively) to determine the size of each webpage element, which in turn depends on the sizes of the elements contained within it.

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

4.4.2 Hardware Design

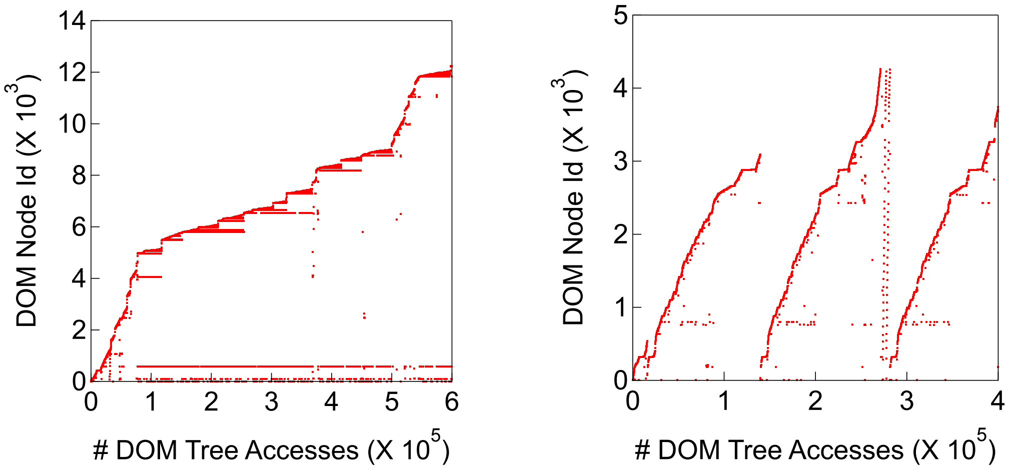
We propose the DOM cache to capture the DOM tree data locality. It sits between the processor and the L1 cache, effectively behaving as an L0 cache. Each cache line contains the entire data for one DOM tree node, which is 698 bytes in our design. Different from the data array in a regular cache, we implement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

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

We choose to implement the DOM cache as a "software-managed" cache–i.e., the data is physically stored in hardware memory, and the software performs the actual cache management, such as insertion and replacement. Prior work has demonstrated effective software-managed cache implementations [102]. It is possible to implement the DOM cache entirely in hardware, similar to a normal data cache. Our motivation for a software-managed cache is to avoid the complexity of a hardware cache. Typically, the cache involves hardware circuitry whose overhead can be high, especially for extremely small cache sizes.

The software overhead for the software-managed browser cache is relatively insignificant for the following reasons. First, a simple replacement policy that always evicts the earliest inserted line is sufficient. Due to the streaming pattern shown in Figure 4.11, DOM tree nodes are rarely re-referenced soon after the browser engine moves past them. Therefore, a simple FIFO design is almost as effective as the least recently used policy, but with much less management overhead.

Second, a very small number of DOM cache entries guarantee a high hit rate. Therefore, the cache-hit lookup overhead is minimal. Figure 4.10b shows how the hit rate changes with the number of entries allocated for the DOM



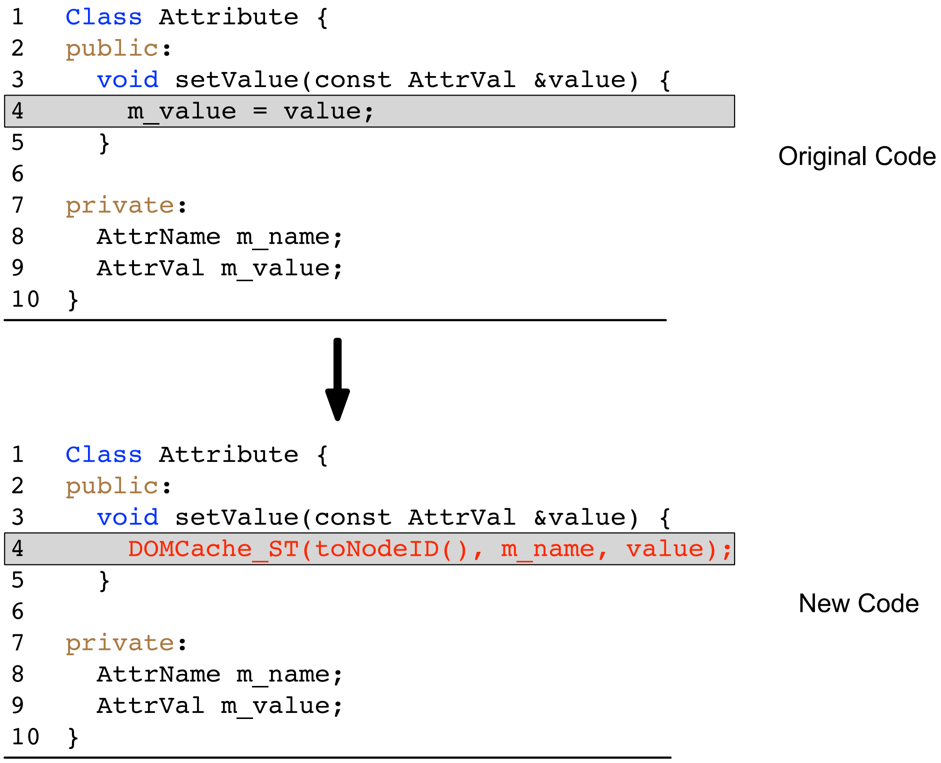
(a) `www.sina.com.cn` (b) `www.slashdot.org`

Fig. 4.11: Representative DOM tree access patterns.

tree. The curve represents the average hit rate, and the error bars represent the standard deviations across different webpages. Across all the webpages, a 4-entry design can achieve about 85% hit rate, and so we use this configuration. In this sense, the DOM cache is effectively a single set, 4-way fully associative cache. Similarly, the render cache contains two entries (i.e., two cache lines). On average, it achieves over 90% hit rate.

4.4.3 Software Support and Programmability

To access a particular DOM tree node in the rendering engine, developers issue `DOMCache_LD(Id, attr)` and `DOMCache_ST(Id, attr, data)` for read and write operation, respectively. Similar APIs are also provided for the Render Cache. In the provided APIs, `Id` represents the DOM tree node ID (similar to the `Style_Apply()` API), `attr` represents a particular DOM node



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

Original Code

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

New Code

Fig. 4.12: Using the `DOMCache_ST()` API in the rendering engine. The new DOM attribute store API (line 4 in the new code) replaces the original attribute value assignment (line 4 in the original code), and performs cache management.

attribute, and `data` indicates the new data of the specified `attr`. Recall that our DOM cache design allows each attribute of a DOM node to be individually addressed (Chapter 4.4.2). The syntax of both APIs allow developers to fully utilize this feature.

Figure 4.12 shows how `DOMCache_ST()` API is used in the rendering engine. It is used to set value of any given attribute in the `setValue()` method of the `Attribute` class. Specifically, `DOMCache_ST()` replaces the original value assignment. The API implementation performs the actual hardware memory accesses as well as cache management, such as replacement and insertion. For example, the API needs to maintain an array, similar to the tag array in a regular cache, to keep track of which DOM nodes are in the cache and whether they are modified. Effectively, the runtime library of DOM cache APIs implements a cache simulator. However, the runtime overhead is negligible due to the simple cache design as described in Chapter 4.4.2.

It is worth noting that using DOM cache APIs only affects the primitive classes of a rendering engine (such as the `Attribute` class in Figure 4.12) while maintaining the interface between primitive classes and the rest of the rendering engine unchanged. For example, rendering engine developers can still use the same `setValue()` method to update an attribute's value. Therefore, we do not expect using the new APIs to affect the development productivity.

4.5 WebCore Evaluation

In this section, we first present the power and timing overhead analysis of the proposed specialization techniques (Chapter 4.5.1). We then evaluate the energy-efficiency implications of the SRU and the browser engine cache individually (Chapter 4.5.2, Chapter 4.5.3). In the end, we show the energy-efficiency improvement combining both customization and specialization (Chapter 4.5.4). In particular, we show that our specializations can achieve significantly better energy efficiency than simply dedicating the same amount of area and power overhead to tune the conventional general-purpose cores.

We evaluate our optimizations against three designs, D1 through D3. D1 refers to the energy-conscious design (P1) that we explored in Figure 4.3. Similarly, D2 refers to the performance-oriented design (P2) in Figure 4.3. D3 mimics the common design configuration of current out-of-order mobile processors. We configure D3 as a three-issue out-of-order core with 32-entry load queue and store queue, 40 ROB entries, and 140 physical registers. It has a 32 KB, 1-cycle latency L1 data and instruction cache, and a 1 MB, 16-cycle latency L2 cache.

4.5.1 Overhead Analysis

We use CACTI v5.3 [169] to estimate the memory structures overhead. We implement the SRU in Verilog and synthesize our design in 28 nm technology using the Synposys toolchain.

Area The size of SRU's scratchpad memory is 1 KB. The DOM cache size is 2,792 bytes. The render cache size is 1,036 bytes. The hardware requirements for the SRU are mainly comparators and MUXes to deal with control flow, and simple adders with constants inputs to compute each CSS property's final value. In total, the area overhead of the memory structures and the SRU logic is about 0.59 mm^2, which is negligible compared to typical mobile SoC size (e.g., Samsung's Exynos 5410 SoC has a total die area size of 122 mm^2 [180]).

Power The synthesis reports that the SRU logic introduces 70 mW total power under typical stimuli. The browser engine cache and the SRU scratchpad memory add 7.2 mW and 2.4 mW to the dynamic power, respectively. They are insignificant compared to power consumption for Web browsing (in our measurements, a single core Cortex-A15 consumes about 1 W for webpage loading). Clocking gating can reduce the power consumption further [129]. But we are conservative in our analysis and do not assume such optimistic benefits.

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

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

4.5.2 Style Resolution Unit

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

Figure 4.13 shows SRU's performance improvement for the *Style* kernel and the entire webpage loading on the performance-oriented design D2 in

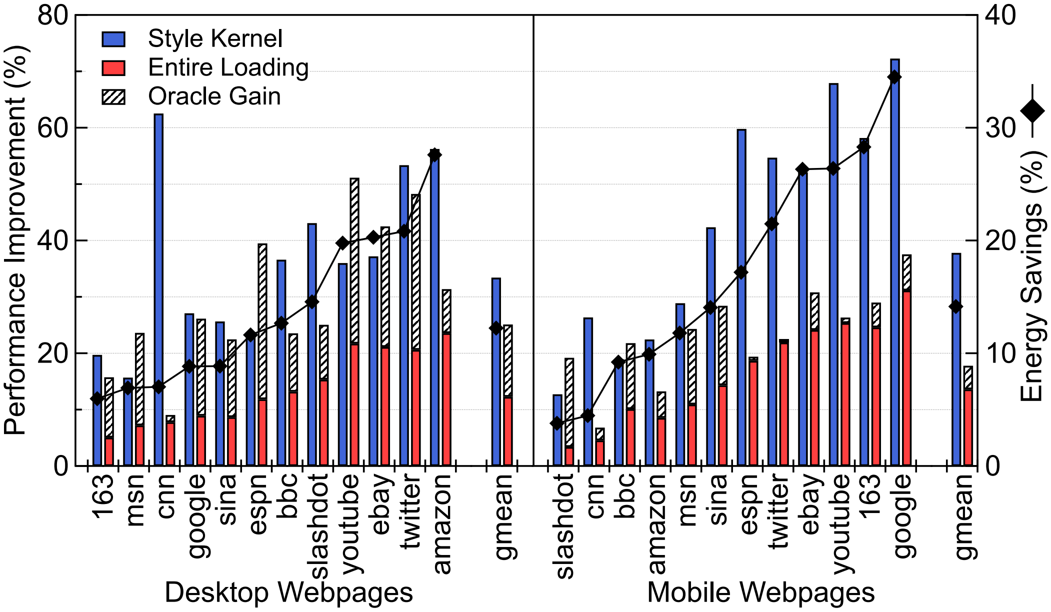


Fig. 4.13: Performance and energy improvement of the SRU.

Figure 4.3. The average performance improvement of the *Style* kernel is 33.4% and 37.8% for desktop and mobile webpages, respectively. Generally, we find that mobile webpages benefit slightly more from the SRU because they tend to be less diversified in webpage styling, and therefore the SRU has higher coverage.

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

The SRU not only improves performance but also reduces energy consumption. The right y-axis of Figure 4.13 shows the energy saving for the entire webpage loading. Webpages are sorted according to the energy savings. On average, SRU results in 13.4% energy saving for all webpages.

Figure 4.13 also shows the oracle improvement if the entire applying phase can be delegated to the SRU (i.e., no hardware resource constraints). Desktop webpages have much higher oracle gain than mobile webpages. The software fall-back mechanism is more frequently triggered in desktop-version webpages due to their diversity in styling webpages. This also implies the potential benefits of reconfiguring the SRU according to different webpages. An SRU that is customized for mobile webpages could potentially be much smaller.

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

4.5.3 Browser Engine Cache

Figure 4.14 shows the energy reduction from using the browser engine cache. The browser engine cache can serve data more energy-efficiently because of the high hit rate of its cache (as shown in Figure 4.10b). Mobile webpages achieve less energy saving than desktop-version webpages because

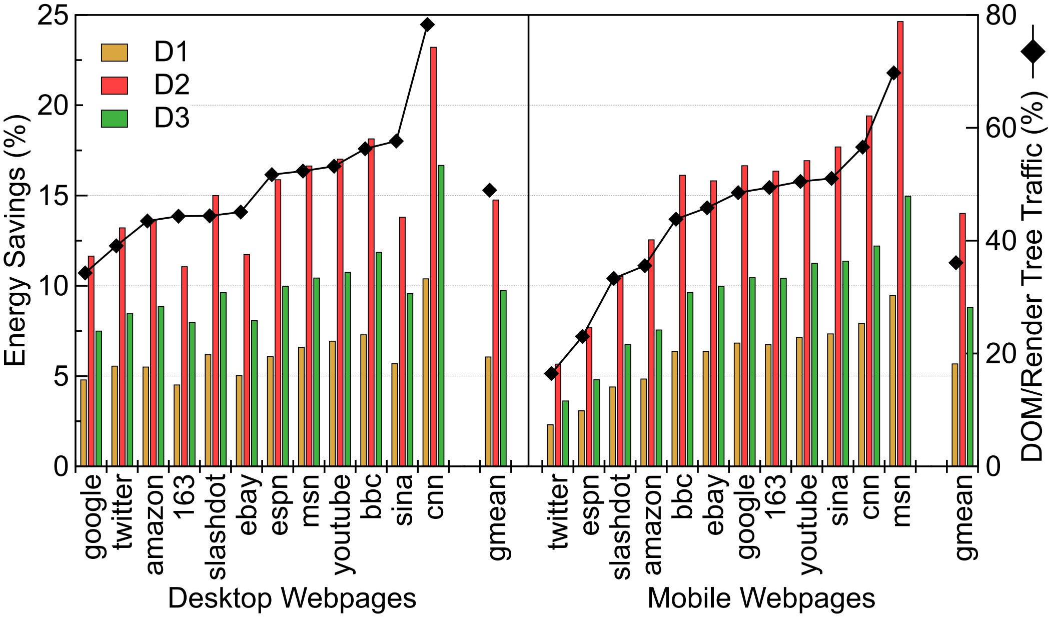


Fig. 4.14: Energy savings with a browser engine cache.

of their smaller memory footprint. On average, the performance-oriented design (D2) achieves 14.4% energy savings. Since the energy-conscious (D1) and current design (D3) have smaller caches, the energy consumption caused by the data cache is less, and therefore benefits less from the browser engine cache. On average, their energy consumption reduces by 5.9% and 9.3%, respectively.

We find that the DOM tree and Render tree access intensity largely determines the amount of energy saving. The right y-axis in Figure 4.14 shows the amount of L1 data cache traffic that is attributed to accessing both data structures. In the most extreme case, about 80% of the data accesses for loading cnn touch the DOM tree and the Render tree. Therefore, it achieves the largest energy saving.

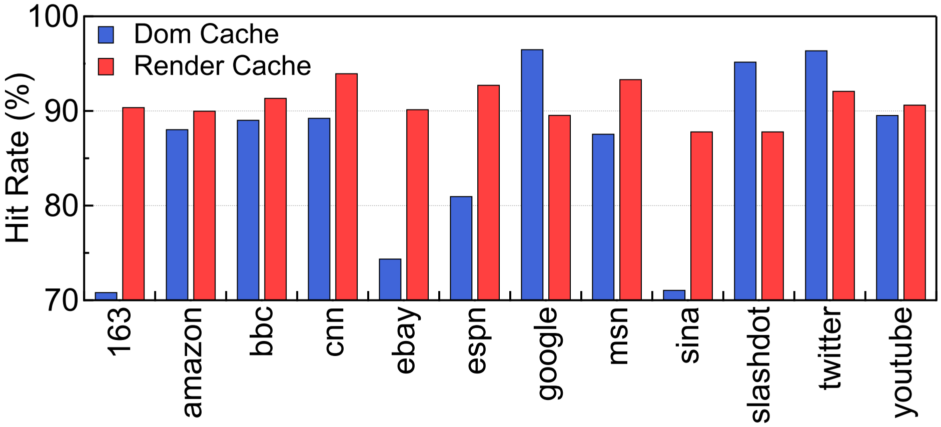


Fig. 4.15: DOM Cache and Render Cache hit rate for desktop webpages.

There are some outliers in desktop webpages where the energy savings are not proportional to DOM/Render tree access intensity. For example, `sina` has a much higher traffic (∼60%) than `twitter` (∼40%), but with similar energy savings. This is because `sina` has a much lower DOM cache hit rate than `twitter`. Figure 4.15 shows the DOM cache and Render cache hit ratio for desktop webpages. We observe that `sina` has a DOM cache hit rate at ∼70%, lower than `twitter` at ∼97%. A lower DOM cache hit ratio indicates the `sina` does not fully use the low-energy browser engine cache. In contrast, we find that mobile webpages all have a high browser engine cache hit rate, and therefore their energy savings closely track the DOM/Render tree traffic.

Due to the software cache management overhead, the browser engine cache incurs performance overhead. Figure 4.16 shows the desktop webpages' execution time of the three designs with the browser engine cache. The values are normalized to each design's baseline configuration without the browser

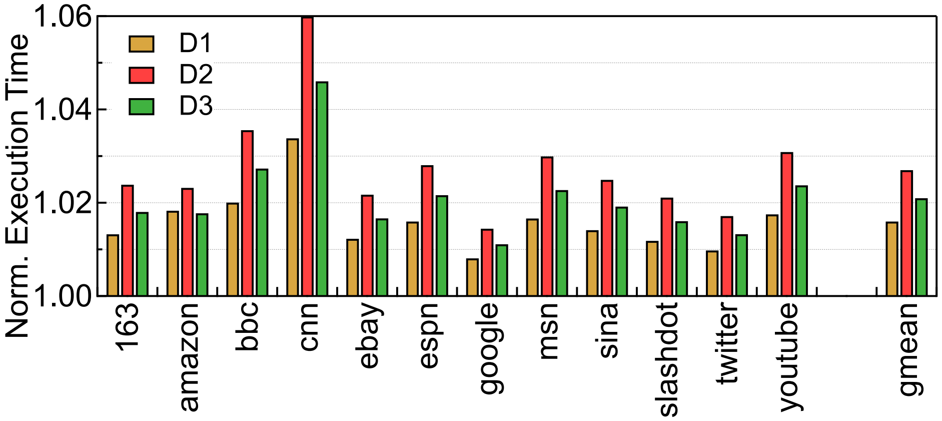


Fig. 4.16: Execution time with the browser engine cache of the three designs. Values are normalized to each design's baseline configuration without the browser engine cache.

engine cache. We find that the performance slow down is minimal, primarily because the design decisions that we made (as described in Chapter 4.4.2) minimize the software management overhead. On average, the slowdown for D2 with a 64 KB L1 data cache is only 2.7%. The slowdown for D1 and D3 with smaller L1 data caches (8 KB and 32 KB, respectively) is slightly smaller– only 1.6% and 2.1%, respectively. We speculate that the reason is that both D1 and D3 have slower performance than D2, and as such, they amortize the overhead of the software cache management.

4.5.4 Combined Evaluation

Figure 4.17 shows the energy-efficiency improvement for the entire webpage loading on all three designs by progressively adding the two optimization techniques. The dotted curve represents the Pareto-optimal frontier of the de-

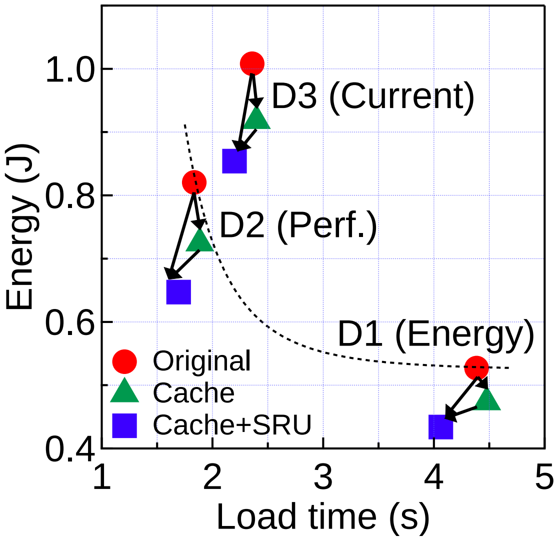


Fig. 4.17: Energy-efficiency improvement over three designs.

sign space discovered in Chapter 4.2.2. The circles represent original designs in this energy-performance space. The triangles represent the new energy-performance trade-off points after applying the software-managed browser engine cache optimization. The squares show the new energy-performance points when the SRU is added atop the caching optimization.

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

After applying the browser engine cache, the performance slightly degrades due to its software management overhead. Therefore, all the triangles move slightly to the right despite the energy savings. However, applying the

SRU optimization improves both performance and energy consumption. All the squares move toward the left corner. In effect, we push the Pareto-optimal frontier in the original design space to a new design frontier with significantly better energy efficiency.

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

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

Our specializations incur area overhead. To quantitatively assess the effectiveness of the area overhead, we compare our results with general-purpose designs that simply use the same area overhead to scale up microarchitecture resources. In our evaluation, we use the additional area to improve the I-cache and D-cache sizes because instruction delivery and data feeding are the two major bottlenecks, as discussed in Chapter 4.2.3. The additional area would be most justified to improve the I-cache and D-cache sizes.

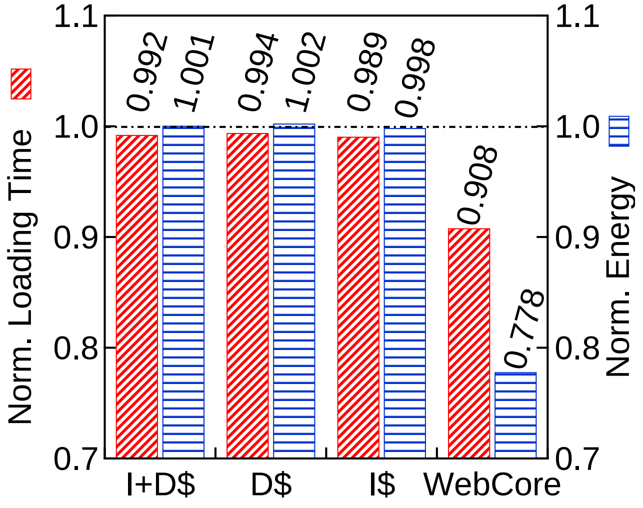


Fig. 4.18: Allocating area for caches versus specializations.

As an example, Figure 4.18 compares our combined specializations (WebCore) with designs that increase the I-cache size by 24 KB (I$), D-cache size by 24 KB (I$), and both caches by 12 KB (I+D$) based on the D2 design. The figure normalizes the webpage loading time and energy consumption to the D2 design without any specializations. We see that simply improving the cache sizes in general-purpose cores achieves only negligible performance improvement (<1%) with a slightly higher energy consumption. However, WebCore specializations provide significantly better energy efficiency.

4.6 Related Work

We first put `WebCore` in the broad context of architecture specialization for Web applications in Chapter 4.6.1. The browser engine cache bears similarities with previous work on specialized cache design, which we discuss in Chapter 4.6.2. Finally, Chapter 4.6.3 discusses prior work on constructing representative mobile Web benchmarks, which is inherently related to our Web application selection process.

4.6.1 Architecture Specializations for the Web

Similar to `WebCore`, SiChrome [63] 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 [76] and ESP [77] 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.

4.6.2 Specialized Cache Design

L0 caches and scratchpad memories [61, 118] have long been used to reduce data communication overhead by acting as small, fast, and energy-conserving data storage. The browser engine cache proposed in this paper demonstrates the effectiveness of such an idea for mobile Web browsing workloads. We propose to implement the browser engine cache as a collection of registers where each register holds exactly one DOM (render) tree attribute. In contrast, the typical L0 cache in mobile SoCs [119] is agnostic to the application-level data structures. Each L0 cache line, thus, holds more than one DOM attribute, leading to excessive energy consumption when accessing individual attributes.

In addition, the strong locality of the principal data structures revealed in our analysis can potentially be captured by dedicating cache ways to the Web browser application [92, 120]. The streaming access pattern of the DOM tree shown in Figure 4.11 indicates that a dynamic cache insertion policy such as DIP [153] or an intelligent linked data structure prefetcher [88] on L1 data cache are also worth exploring. However, the browser engine cache we propose aims at saving energy with minimal loss in performance, which the prior performance-oriented techniques have not been proven/claimed to provide.

4.6.3 Web Applications Characterization

BBench [101] is a webpage benchmark suite that includes 11 hot webpages. Its authors perform microarchitectural characterizations of webpage loading on an existing ARM system. Although the authors show that the 11 webpages have distinctly different characteristics from SPEC CPU 2006, they do not quantify the comprehensiveness and representativeness of the webpages against the vast number of webpages "in the wild." In stark contrast, our analysis in Chapter 4.1 systematically proves the broad coverage of our webpages, which is needed for robustly evaluating the impact of the optimizations that we propose. For example, we find that BBench does not include significantly complex webpages, and our analysis led to including two webpages of that sort, i.e., `www.163.com` and `www.sina.com.cn`. Their webpage sizes are about 4x larger than the average BBench webpage, and as such are needed to increase the coverage of our benchmarking suite.

MobileBench [148] characterizes the performance impact of various microarchitecture features on mobile workloads. Our paper quantifies the performance-energy trade-off, and focuses specifically on Web applications. Complementary to our design space exploration, MobileBench results show that more aggressive customizations of other microarchitecture structures such as the prefetcher are worth exploring.

Chapter 5

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 [124] (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 [124, 167] 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 [20, 44]. I quantitatively show that Web applications particularly benefit from the heterogeneity offered by the ACMP architecture to achieve an ideal balance between QoS experience and energy consumption.

Leveraging the ACMP architecture, I propose `WebRT`, a Web runtime that minimizes energy while guaranteeing satisfactory user QoS experience by scheduling Web application executions using proper ACMP configurations. The key insight is that we must devise different optimization schemes according to the nature of different user interaction forms. To that end, I introduce a user-application interaction model called LTM. LTM captures three fundamental user interaction forms in mobile Web applications–Loading, Tapping, and Moving–and provides a framework for reasoning about different energy optimization strategies.

The rest of this chapter is organized as follows. Chapter 5.1 presents the hardware and software experimental methodology. Chapter 5.2 introduces the LTM interaction model and points out that the runtime mechanisms need to be different for different interaction forms. Using loading (L) as a case study, Chapter 5.3 quantitatively shows that an ACMP is beneficial for mobile Web and therefore is a natural candidate for `WebRT`. Chapter 5.4 describes the `WebRT` components for L, and Chapter 5.5 describes the `WebRT` component for T and M. Finally, Chapter 5.6 compares the contrasts `WebRT` with prior work on software support for mobile Web.

5.1 Experimental Setup

Software Infrastructure `WebRT`-related experiments and implementations are performed on Google's open-source Chromium browser engine, which is used directly in the Chrome browser and is the core of many other popular browsers, such as Opera and Android's default browser. We use Chromium version 48.0.2549.0, which is the most recent version at the time of my work. The modified Chromium runs on unmodified Android version 4.2.2.

Hardware Platform We use the ODroid XU+E development board [107], which contains an Exynos 5410 SoC that is known for powering the Samsung Galaxy S4. The Exynos 5410 SoC contains a representative ACMP architecture comprising an energy-hungry high-performance (big) core cluster and an energy-conserving low-performance (little) core cluster. The big and little clusters can be individually disabled and enabled. The big cores are ARM Cortex-A15 processors that operate between 800 MHz and 1.8 GHz at a 100 MHz granularity. The little cores are ARM Cortex-A7 processors that operate between 350 MHz and 600 MHz at a 50 MHz granularity. The frequency switching and core migration overhead is 100 μs and 20 μs, respectively [183, 185].

Energy Measurement `WebRT` focuses on the processor power consumption because the processor power has been steadily increasing and has gradually become the most significant power consumer in a mobile device compared to other components such as the screen and radio (Chapter 3.2).

We measure the processor power and energy consumption on real hard-

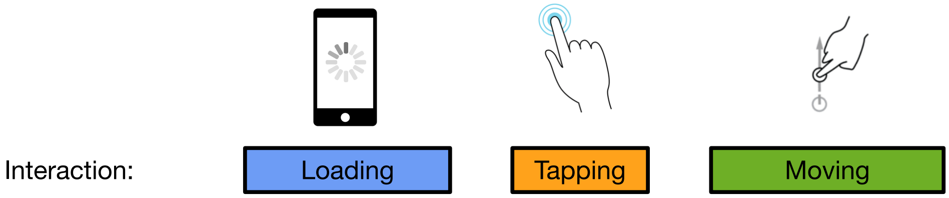


Fig. 5.1: The LTM (Loading-Tapping-Moving) user-application interaction model of mobile Web. LTM captures three primitive types of interaction: page loading, finger tapping, and finger moving. We use LTM as a framework to reason about user QoS experience.

ware as follows. The ODroid XU+E development board has built-in current sense resistors (10 mΩ) for both the big and little cores. We use a National Instrument DAQ Unit X-series 6366 to collect voltage measurements at these sense resistors for the big and small CPU clusters at a rate of 1,000 samples per second, and thereby derive the power consumption. Energy consumption is computed by multiplying power with real execution time.

Reproducibility We repeat every experiment that we study 3 times. Unless otherwise mentioned, the results we report are the median of all runs. We find the run-to-run variations are usually about 5%, and do not affect our conclusions. We use Mosaic [103], a UI-level record and replay tool, to ensure consistent user interaction and to reduce human-induced noise across different runs on the same application.

5.2 LTM Model of Mobile User Interaction

To systematically analyze user interactions in mobile Web applications, we introduce a simple conceptual model called LTM, which captures three primitive user interaction forms in mobile Web applications: loading application page (L), tapping the display (T), and moving finger on the display (M). Figure 5.1 illustrates the LTM model.

The three interactions cover a majority of human-computer interactions on mobile devices. This is because 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 moving interaction in particular could be manifested in various ways, such as scrolling, swiping, or even drawing a picture. Internally, each user interaction is translated to one or more application event. For example, a tapping interaction is often translated to a `touchstart` and a `touchend` event, and a moving interaction can be translated to a `scroll` event or a `touchmove` event depending on context. In this paper, we focus on the following events that could be triggered by LTM interactions on a mobile device: `click`, `scroll`, `touchstart`, `touchend`, and `touchmove`. We do not consider events specific to desktops (e.g., `drag`, `mouseover`) that are generally not fired on mobile devices.

The runtime optimization strategy for Loading is 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 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 will discuss the `WebRT` component that targets the Loading in Chapter 5.4 and the component that targets Touching and Moving interactions in Chapter 5.5 separately.

5.3 Motivation: Energy-Delay Trade-off

An ACMP consists of cores with different computation capabilities–often with different microarchitectures, such as big out-of-order cores and small in-order cores. Each core has a variety of frequency settings. Different core and frequency combinations provide a wide range performance and energy characteristics. The flexibility of an ACMP architecture to make trade-offs between performance and energy consumption leads us to answer a fundamental question: do Web applications benefit from an ACMP heterogeneous systems? For example, can a processor lower the frequency for a simple webpage to consume less energy but still respect the QoS deadline? Can a webpage originally scheduled on an energy-consuming core be migrated to an energy-saving core without violating the QoS constraint?

We quantitatively answer this question using webpage loading as a case study. The same experimental methodology and conclusion also hold true for the other two types of interactions in mobile Web applications. We base our measurements and analysis on the 5,000 hottest webpages on the Internet ranked by `http://www.alexa.com/`. We show that different webpages require different core and frequency configurations to meet a given deadline of webpage loading while minimizing the energy. This suggests that ACMPs with both big and smalls core, each capable of performing DVFS, are strongly beneficial.

To demonstrate the benefits of such heterogeneous systems, we measure the webpage load time and energy consumption of the 5,000 webpages on the Cortex-A9 and A8 processors. We sweep a total of seven configurations available on the big and little cores, i.e., Cortex-A9 with four DVFS settings and A8 with three DVFS settings, respectively. We begin our analysis with four webpages that represent the general trends that we observe (Chapter 5.3.1), and we subsequently expand our analysis to include the comprehensive set of all webpages (Chapter 5.3.2).

5.3.1 Representative Analysis

Figure 5.2 shows the energy versus delay plots for the four representative webpages. Assuming 3 seconds as the cut-off latency for webpage load [32], the four webpages have different ideal core and frequency configurations to meet the cut-off while simultaneously minimizing the energy consumption. For example, `www.autoblog.com` is a complex website that has 4,235 nodes in the DOM tree, and it therefore requires the highest frequency on the big core to meet the cut-off latency. However, this configuration is overpumped for simpler websites such as `www.newegg.com` with 3,152 DOM tree nodes. It only requires 700 MHz of the big core. This suggests that some webpages can benefit from different frequencies in each processor's core.

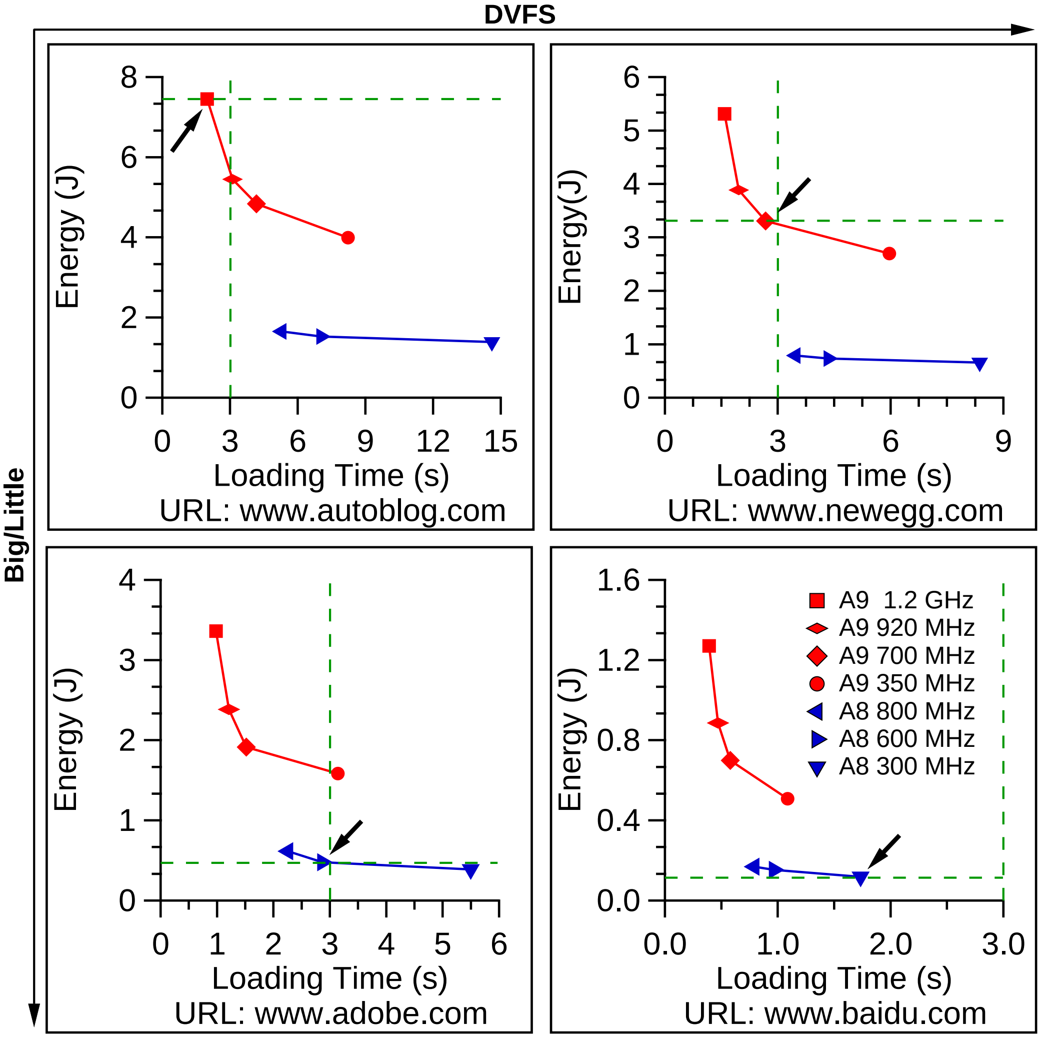


Fig. 5.2: Webpages have different ideal execution configurations to meet the cut-off latency while consuming the least energy.

In addition, some webpages can take advantage of scheduling between big/little cores. If only the big core is available, `www.adobe.com` can at best be loaded at 700 MHz. Instead, with the little core, the webpage can be loaded using 600 MHz, which still meets the cut-off latency but consumes 75% less energy than 700 MHz on the big core. Similarly, `www.baidu.com` is a search engine website that has very concise content with less than 1 KB of images. It only requires the lowest frequency on the little core.

5.3.2 Comprehensive Analysis

We extend our analysis to the full set of 5,000 webpages. Figure 5.3 shows the distribution of ideal core and frequency configurations for different cut-off latencies, ranging from 1 second to 10 seconds at 1 second intervals. Each region in Figure 5.3 represents the portion of webpages that are loaded at the corresponding architectural configuration with minimal energy consumption while still meeting the cut-off latency. We find a wide distribution of ideal configurations, indicating the benefits of a flexible baseline architecture that mixes big/little cores with different frequencies.

Assuming a tight 3 second cut-off latency [32], a single core with a fixed frequency is insufficient for a wide spectrum of webpages. The best single core with a fixed frequency is the little core with 600 MHz. However, it can only load 40.2% of the webpages within that latency constraint. Even a single core (big or little) with varying frequencies is insufficient. When we consider the little core with varying frequencies, only 74.4% of webpages can

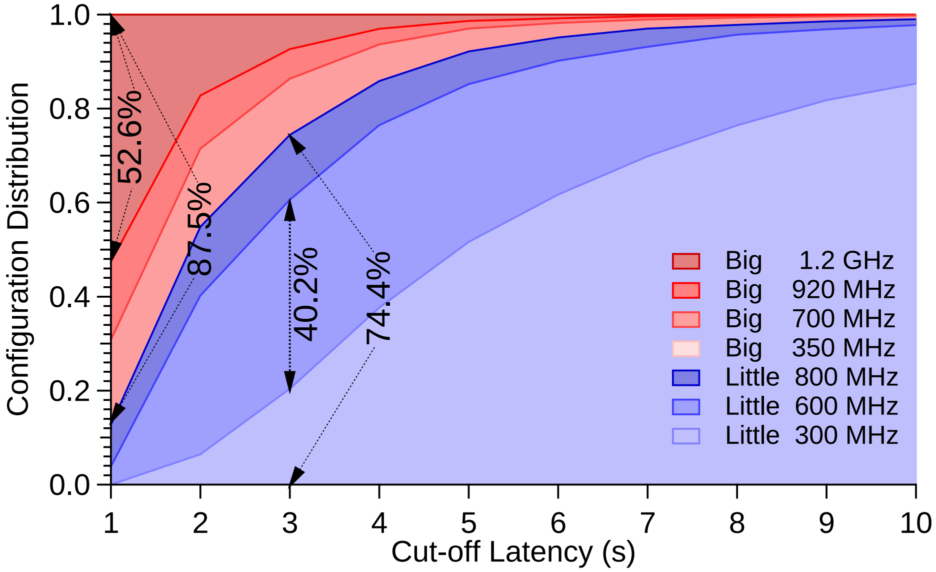


Fig. 5.3: The distribution of ideal core and frequency configurations under different cut-off latencies.

be loaded within the cut-off latency. However, if we use a big core to load all the webpages, then the 74.4% of webpages have suboptimal performance-energy trade-off. Furthermore, a simple heterogeneous system with both a big and little core but each with a fixed frequency may also cause suboptimal performance-energy trade-off for some webpages. Statistically, the best single-frequency configurations are 700 MHz on the big core and 600 MHz on the little core; yet, a heterogeneous system with only these two settings leads to ideal scheduling for only 52.1% of the webpages.

Although 3 seconds is the typical cut-off latency on mobile systems, we also study the sensitivity of the ideal configuration distribution under other cut-off latencies. We find that varying cut-off demands also call for a flexible baseline architecture. As Figure 5.3 shows, no one particular configuration consistently performs well under varying cut-off latency requirements. For example, although relaxed cut-offs favor the little core, it is suboptimal for 87.5% of the webpages under a tight 1 second constraint. Similarly, the big core, which performs very well under tight cut-offs, is overpumped under more relaxed constraints; it is only needed for about 3% of webpages when the cut-off latency is 10 seconds.

In summary, we find that different webpages require different ideal core and frequency settings to achieve the ideal balance between performance and energy-efficiency. Varying cut-off latencies also demand different ideal configurations. Therefore, we conclude that different webpages can strongly benefit from a versatile heterogeneous system consisting of both big and little cores each capable of performing DVFS.

5.4 Webpage-aware Scheduling

In this section, I first show that it is possible to predict the webpage load time and energy consumption using merely webpage-inherent characteristics (Chapter 5.4.1). 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. We quantitatively show that our predictive models achieve a

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 5.1: Model Predictors

desirable accuracy (Chapter 5.4.2).

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 (Chapter 5.4.3). Real hardware and software measurements show that against a performance-oriented hardware strategy, the webpage-aware scheduler achieves 83.0% energy savings while violating the cut-off latency for only 4.1% more webpages. Compared with a more intelligent, on-demand OS DVFS scheduler, the mechanism achieves an additional 8.6% energy savings along with a 4.0% performance improvement (Chapter 5.4.4).

5.4.1 Performance and Energy Modeling

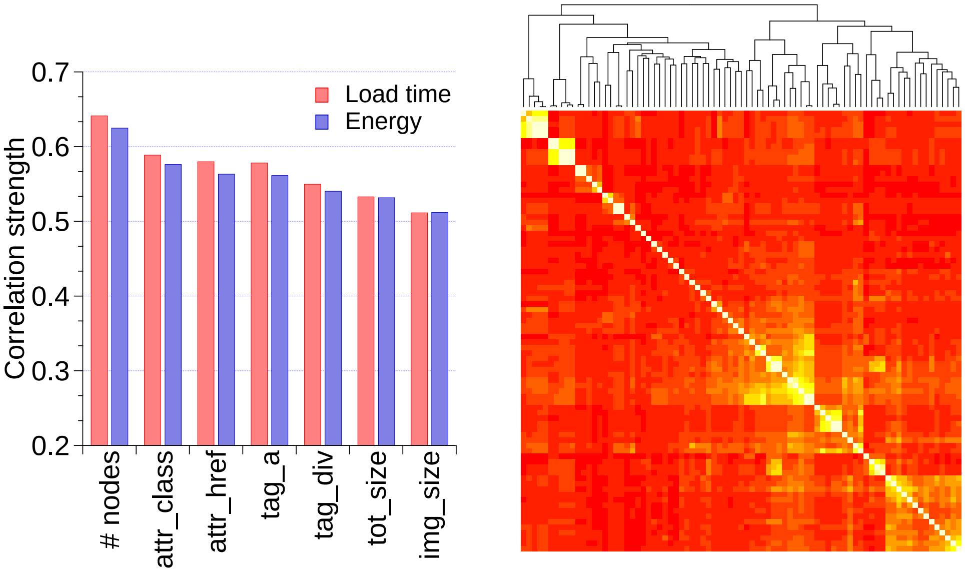
Model Derivation We find that regression models provide sufficient accuracy to predict the webpage load time and energy consumption. A regression model is a mathematical function between a set of predictors and a response. Within our context, the response is either the webpage's load time or energy consumption in loading the webpage. The predictors are a set of webpage characteristics. 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) [109].

We consider two types of predictors. The first type includes the *webpage-inherent* primitives such as the number of HTML tags. These primitives have show strong inter-webpage differences, and as such have a strong influence on the load time and energy consumption. 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. A media website is a classic example where content-dependent characteristics are dominant. For example, a news website, such as `www.bbc.com`, has a relatively stable appearance. Its structural layout (i.e., HTML) and style (i.e., CSS) do not change frequently. However, the website's content is changing daily to keep up with the latest breaking news. For instance, they are constantly updating images, and image sizes have a significant impact on the webpage load time. In our measurement we observe a 4X load time difference between a 200 KB and 50 KB image. Therefore, it is necessary to consider both webpage-primitive and content-dependent characteristics for modeling the load time and energy consumption of webpage load.

We summarize these features in Table 5.1. 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.

Model Specification and Refinement We apply various techniques to mitigate overfitting and capture predictor-response nonlinearity to achieve high prediction accuracy. We use R [31] and its *glmnet* and *rms* packages for all analysis.

We consider a large number of predictors (376) relative to the number of observations (2,500). This is known to produce predictions that result in overfitting [109]. We mitigate this, to the first order, by eliminating predictors that are less correlated to the response. We test the predictor/response correlation strength by calculating the squared correlation coefficient (ρ^2) between each predictor variable and observed load time and energy. Figure 5.4a shows the seven most-correlated predictors. For both load time and energy, we



(a) Predictor-response correlation.

(b) Predictor self-correlation.

Fig. 5.4: Predictor correlations.

find the number of DOM tree nodes (#nodes) is the most-correlated webpage primitive because it heuristically captures the webpage structure's complexity. Also, both image size and the total webpage size are also correlated because they capture the webpage content. We only select predictors with ρ^2 greater than 0.01.

We further minimize overfitting by pruning features that are correlated to each other. We test the correlation across predictors left after predictor strength test. The correlation matrix is shown as a heatmap in Figure 5.4b. The intensity of a point in the heatmap is proportional to the magnitude of the correlation coefficient between two predictors. The height of the branches in the dendrogram quantifies this magnitude.

In general, we find two types of correlation: inherent correlation and imposed correlation. Several HTML tags and attributes are functionally defined symbiotically and most often used together, exemplifying the inherent correlation. For example, the <`form`> tag describes a form in the webpage, and the `action` attribute specifies where to submit the form. These two predictors are almost synchronized with each other, suggesting redundancy. Similar examples are the <`a`> tag and the `href` attributes, which are defined to specify an external hypertext link. Some other predictors do not bear such an inherent relationship, but web developers use them together to describe related information, such as an image's width and height. For example, CSS properties `height` and `width` are highly correlated. The descendant selector pattern and class selector pattern also show heavy correlation for this reason.

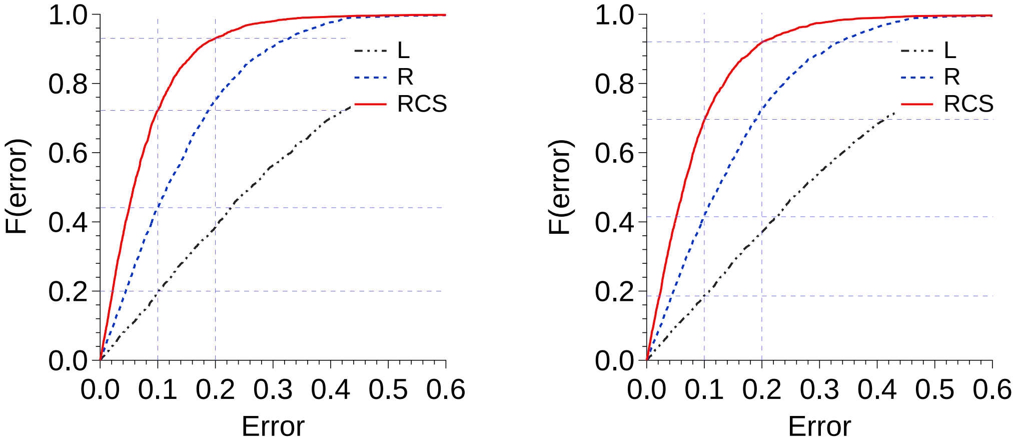
Furthermore, it is unlikely that the true relationship between the response and all predictors is strictly linear as assumed by simple linear models. One effective method to model nonlinearity is to fit data with *restricted spline* functions that are piecewise polynomial functions but which force linear fitting beyond the first and last knots [109].

5.4.2 Model Evaluation

To validate the model, we obtain 2,500 observations in addition to the 2,500 observations used for deriving the model. We incrementally evaluate the effect of various refinement techniques described previously by comparing the accuracy of three regression models. First, we evaluate a basic linear regression (L) model that prunes less-significant predictors. Second, we evaluate linear regression with regularization (R) that further prunes predictors correlated with each other. Third, we evaluate a restricted cubic spline-based (RCS) model using pruned features, which captures the nonlinear relationship between predictors and responses. Of all three models, RCS performs best at predicting both load time and energy. We show all three models for completeness of evaluation.

Performance model The basic linear regression model (L) has a median and mean error rate of 25.8% and 32.8%, respectively, indicating a less-desirable prediction. The regularization-based model (R) reduces the median and mean error rate to 11.5% and 13.6%, respectively, due to more aggressive predictor pruning. Restricted cubic spline (RCS) modeling predicts the best, with the median and mean error rate of only 5.7% and 7.5% due to its capability of capturing more complex relationships between predictors and responses.

We also assess the distribution or prediction errors. Figure 5.5a shows the results by presenting the cumulative distribution of the error for three modeling methods. Each (x, y) point in the graph corresponds to the portion of pages (y) that are at or below a particular error rate (x). Owing to overfitting, L predicts very accurately for a few webpages, but lacks the capability to be generally applicable to a large range of webpages. As a result, L can only predict 20.0% of the webpages within 10% error. In contrast, R mitigates overfitting due to aggressive pruning, and predicts 44.6% of the webpages



(a) Load time model.

(b) Energy model.

Fig. 5.5: CDF of prediction errors.

within 10% error. Finally, RCS further captures the nonlinear relationship, and therefore can predict 73.0% of the webpages within 10% error, and 94.0% webpages within 20% error.

Energy Model Similar to the load time model, the RCS-based model performs the best, with the median error rate of 6.4% (mean of 8.2%), dropping from the median of 12.3% and 27.1% for R and L, respectively. Figure 5.5b shows the cumulative distribution of the error for three modeling methods. For reasons explained earlier, RCS can predict 70.0% of the webpages within 10% error (91.8% within 20% error), improving from 41.7% and 18.7% of R and L, respectively.

5.4.3 Scheduler Implementation

Scheduler During the parsing stage, which takes <1% of the total execution time, 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.

Scheduling Overhead We consider two major scheduling overheads: prediction and configuration transitioning. Prediction occurs very rapidly (<3 milliseconds on the Cortex-A9 under 1.2 GHz). Moreover, prediction is interleaved with the parsing stage of the rendering engine. As parsing in modern browsers is highly optimized (e.g., asynchronous with the other processing), the prediction overhead is insignificant. On the basis of our measurements, we assume a constant overhead of 5 milliseconds.

Transitioning between hardware configurations involves the penalty of migrating tasks between big/little cores and/or frequency scaling overhead. The major overhead source of task migration is context switch, i.e. (re)storing architecture state such as register files and configuration registers, as well as warming up the private L1/L2 caches (assuming cache coherency between the last-level cache (LLC) of big and little cores). We assume a constant overhead of 20 milliseconds for state (re)storing per context switch, as indicated for the ARM big.LITTLE system [43]. For private cache warmup penalty, prior work shows that performance often improves when private LLCs of big and little cores are powered on together [82]. Thus, we ignore the warmup penalty. Also, prior work suggested that the power overhead of task migration is <0.75% [154]. Thus, we do not consider the additional energy consumption of our scheduling mechanism.

For frequency scaling, we assume 0.3 milliseconds as the overhead. The Linux kernel uses this value on both the Cortex-A9 and A8 systems. This value takes into account both hardware (i.e., voltage regulator module switching frequency) and software overhead (i.e., privilege-level switching overhead for the frequency change request). In our evaluation, since we do not know which configuration the web browser is currently running in, we conservatively consider both the configuration transitioning overhead and the frequency scaling overhead at every scheduling point.

5.4.4 Evaluation

Baseline Mechanism We compare the webpage-aware scheduling mechanism against an intelligent synthesized OS scheduler that performs on-demand DVFS on a heterogeneous system. The OS scheduler scales the frequency during a webpage load based on simple heuristics of system utilization [99, 175]. It samples the CPU usage at a certain period and scales up the frequency if the average CPU usage in the previous sampling period is above a preset threshold, and vice versa. Because no Linux scheduler can yet perform

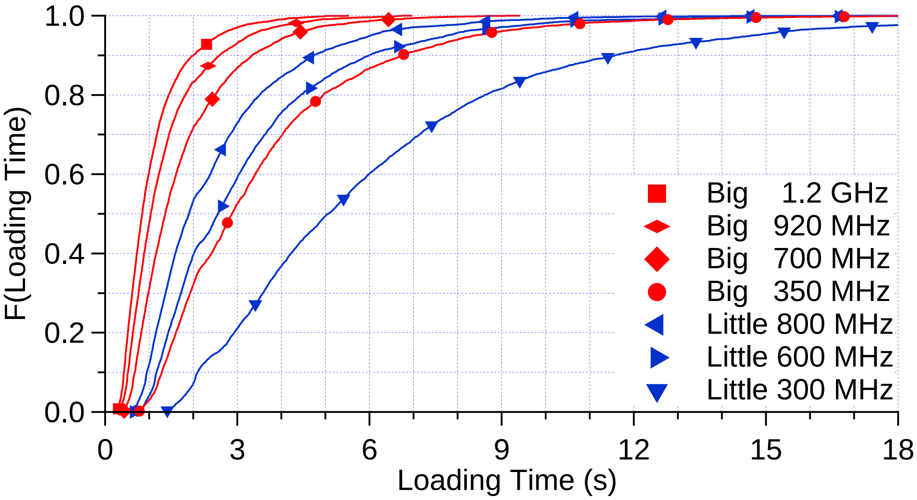
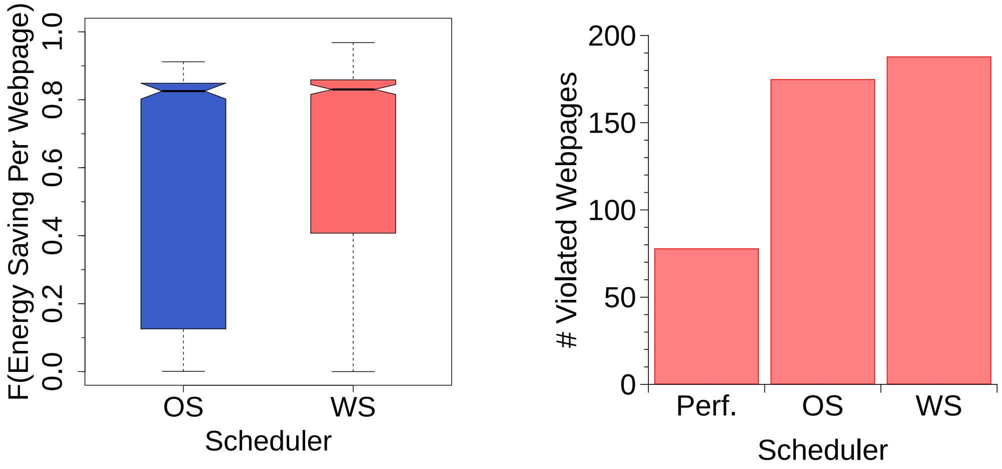


Fig. 5.6: CDF of webpage load time under different configurations.

heterogeneous scheduling across big/little cores, we synthesize such a scheduler by running the webpages under the "on-demand" cpufreq-governor [147] on the big core and the little core, individually, and then choose the better result.

We compare the two scheduling techniques with a baseline strategy that consistently yields the best performance. We determine such a baseline by assessing the performance of all the different core and frequency configurations. Figure 5.6 shows the cumulative distribution of webpage load time under each configuration. Each (x, y) point in the figure represents the portion of webpages (y) loaded within a certain delay (x). The big core with the peak frequency (1.2 GHz) achieves the best overall performance. It can load 96.5% of the webpages within 3 seconds. As the frequency and core capability degrade, fewer webpages can be loaded within the same cut-off latency.



(a) Distribution of per webpage energy saving against the baseline.

(b) Number of webpages that load under the strict cut-off latency of 3 seconds.

Fig. 5.7: Evaluation of different scheduling strategies.

Therefore, we choose the big core (A9) with its peak frequency (1.2 GHz) as the high-performance baseline.

Energy savings We evaluate the same 2,500 webpages that we used to assess the accuracy of the regression models. Assuming a 3 second cut-off latency, Figure 5.7a shows the boxplot of per-webpage energy savings under the webpage-aware and OS schedulers against the high-performance mode. Both schedulers achieve significant energy savings over the high-performance baseline, with a (geometric) average of 83.6% and 83.0%, respectively. This is because both schedulers can schedule webpages to the lower power core or lower frequency.

The webpage-aware scheduler has a denser energy-saving distribution toward 100% than the OS scheduler. This indicates that generally the webpage-aware scheduler achieves higher energy savings. Figure 5.8a shows the histogram of per-webpage relative energy of the webpage-aware scheduler to the OS scheduler. The webpage-aware scheduler saves energy for about 80% of the webpages. There are several webpages that are mis-scheduled onto the big core that could have met the cut-off latency with the little core. These webpages consume much higher energy under the webpage-aware scheduler than the OS scheduler (>2X in Figure 5.8a). On average, the webpage-aware scheduler reduces energy consumption by 8.6% compared with the OS scheduler.

Performance impact Both the OS scheduler and the webpage-aware scheduler trade performance for better energy savings compared with the performance mode. We evaluate their behaviors more critically using the number of webpages that violate the cut-off latency under their operations. This data is shown in Figure 5.7b. The performance mode violates only 3.5% of the webpages with a 3 second cut-off latency because it always operates at peak computational capability. Both of the software schedulers perform slightly worse. Our mechanism, the webpage-aware scheduler, results in 7.6% violations, which is only 0.6% worse than the OS scheduler. However, on (geometric) average, our mechanism loads webpages 4.0% faster than the OS scheduler.

Cut-off sensitivity To assess the webpage-aware scheduler under variable user demands and mobile device conditions, we also experiment with different cut-off latencies. For example, when the end user requests faster webpage load at 2 seconds, the mechanism achieves 7.3% energy savings over the

OS scheduler while violating 4% fewer webpages. In a battery conservation mode where performance is less critical and the cut-off latency is relaxed to 10 seconds, the webpage-aware scheduler achieves 11.8% energy savings compared with the OS scheduler while exceeding the cut-off latency for only 0.02% webpages in total. We conclude that the webpage-aware scheduler is flexible to changing user requirements.

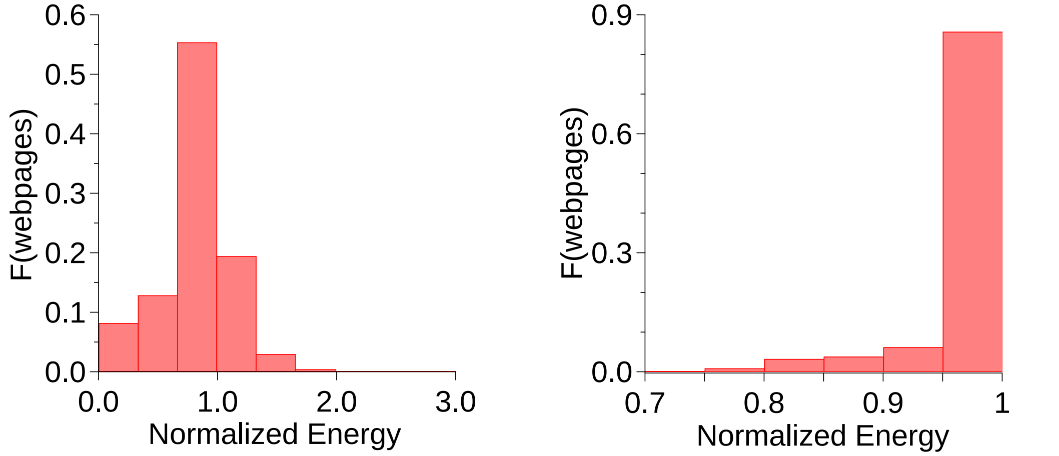
Prediction Accuracy Scheduling effectiveness relies on the load time and energy prediction accuracy. We study the impact of the prediction accuracy by comparing webpage-aware scheduling with an Oracle scheduler that assumes perfect prediction under the 3 second cut-off latency. There are two types of misprediction: over-prediction causes webpages to load on a more powerful configuration that consumes more energy than the ideal one but does not cause cut-off violation; under-prediction loads webpages on a weaker configuration that consumes less energy but violates the cut-off constraint. Our models lead to 10% over-prediction and 4.1% under-prediction. Compared with the Oracle scheduler, the webpage-aware scheduler results in 4.1% cut-off violation but “conserves” 9.7% energy.

Analysis The advantage of the webpage-aware scheduler lies in its awareness of the webpages characteristics and the cut-off latency. As a result, it predicts and chooses a proper, albeit fixed, configuration for each webpage. In contrast, the OS scheduler’s DVFS decision is based on the system utilization, which has no direct correlation with the webpage characteristics/cut-off latency and is sensitive to other system activities. Therefore, it may lead to a suboptimal performance-energy trade-off or even miss the cut-off constraint.

For example, when loading `www.newegg.com` (top-right in Figure 5.2) under the OS scheduler, we find that the CPU usage on the big core reaches above 95% for around 40% of the time and (unnecessarily) incurs peak frequency (i.e. 1.2 GHz). When in fact, the big core with 720 MHz chosen by the webpage-aware scheduler is sufficient to meet the 3-second cut-off latency, achieving 20% energy savings compared with the OS scheduler in our experiments.

However, the flexibility to scale the frequency while loading a webpage sometimes allows the OS scheduler to exploit the marginal value of energy, i.e. a slight increase in energy (through frequency scaling) can bring the webpage back within the cut-off latency that would have been missed if the webpage were loaded using a lower frequency.

For example, `www.autoblog.com` (top-left in Figure 5.2) when loaded under 920 MHz (on the big core) just surpasses the 3-second deadline by 0.1 seconds, but has to fall back using 1.2 GHz under the webpage-aware scheduler. At 1.2 GHz, the webpage loads in only 1.8 seconds but consumes 37% more energy than 920 MHz. However, under the OS scheduler, our statistics show that the OS boosts the frequency above 920 MHz for only around 20% of the time, and finishes the load in 2.7 seconds. Compared with the webpage-aware scheduler that runs at 1.2 GHz for this webpage, the OS scheduler in this case saves 20% energy, effectively exploiting the high marginal value of energy.



(a) Relative energy of the webpage-aware scheduler against the OS scheduler.

(b) Relative energy of the integrated scheduler against the webpage-aware scheduler.

Fig. 5.8: Distribution of per-webpage energy comparisons.

Integrated Scheduler For complete evaluation, we also assess an integrated scheduler that combines the webpage-aware scheduler with OS DVFS. The purpose is to exploit the potentially high marginal value of energy via OS DVFS, but bound the DVFS space to avoid frequencies that are unnecessarily high (wasting energy) or low (missing the cut-off latency).

Specifically, the webpage-aware scheduler first restricts the OS DVFS scheduling space to two frequencies: a lower frequency that just meets the cut-off constraint and a upper frequency that just misses the constraint. Given the two frequencies, the webpage-aware scheduler tries to ensure that the cut-off latency can still be met by further tuning the percentage of time spent in either frequency. In practice, we set the scaling_max_freq and scaling_min_freq of the Linux cpufreq-governor to the lower and upper frequency, respectively. We set the up_threshold to control when to promote to the higher frequency [147]. For example, for `www.autoblog.com` (top-left in Figure 5.2), the OS DVFS on the big core would only operate on 1.2 GHz and 920 MHz. Because 920 MHz is nearly able to hit the deadline, only a small portion of the webpage load must be run in the upper frequency.

Figure 5.8b shows, under a 3 seconds cut-off constraints, the histogram of per webpage relative energy of the integrated scheduler to the webpage-aware scheduler. The integrated scheduler consistently out-performs the webpage-aware scheduler with 3.0% average energy savings (up to 30%). We leave the full integration and detailed comparison for future work.

5.5 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. We first present our motivation for performing event-based scheduling at the event handler level (Chapter 5.5.1). We then provide a high-level design overview of the event-based scheduling framework (Chapter 5.5.2) and then describe its implementation details (Chapter 5.5.3).

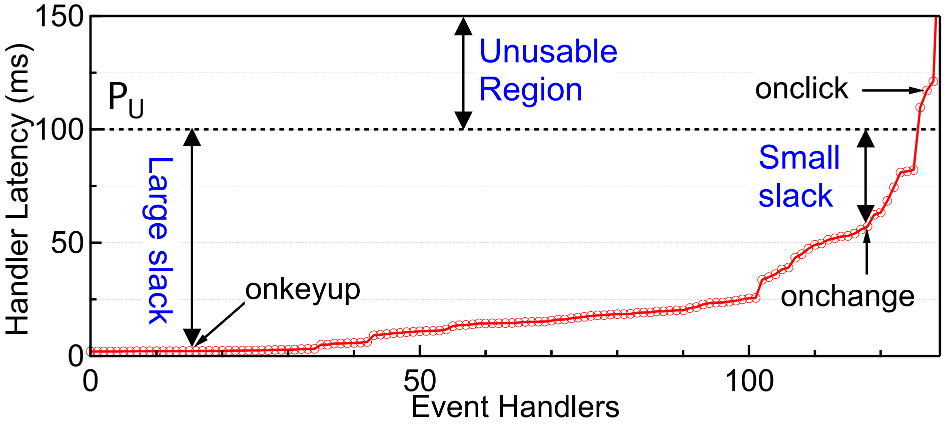


Fig. 5.9: Event handler variation in Ember.js todo list application.

5.5.1 Scheduling Unit

The scheduling unit in the event-based scheduler is the event handler. Whenever an event is triggered, a corresponding event handler is executed. Figure 5.10 provides an example, showing how event handlers H1, H2, and H3 (in that order) are pushed into the event queue for execution. For events that share the same performance constriant, we find that their event handlers have different execution latencies, and therefore lead to different performance slacks. We must treat each event handler differently and make scheduling decisions at that granularity.

We explain the variation in the event handlers' execution behavior using the Ember.js-based todo list application. Figure 5.9 shows the sorted execution latencies of all the event handlers. The x-axis corresponds to the event handlers and the y-axis corresponds to the event handlers' execution latencies. In this example, we assume that the performance target for the scheduler is 100 ms, which is a common performance target for a smooth responsiveness.

We observe a large latency variation for the handlers in Figure 5.9. We label three of the application's representative event handlers as the application executes: `onkeyup`, `onchange`, and `onclick`. The `keyup` event handler only processes one keystroke and therefore finishes execution very quickly in just 2 ms, which leaves a large amount of slack (98%) for the scheduler to exploit. In contrast, the `onchange` event handler adds one entry into the todo list. It requires about 50 ms for execution, which translates to only about 50% slack in performance. Lastly, the `onclick` event handler deletes all the entries in the todo list. The processing time exceeds the performance constraint, and as such there is no opportunity to exploit performance slack. Instead, it requires a higher performance configuration, if available.

5.5.2 Scheduler Design Overview

The event-based scheduler predicts the ideal heterogeneous architecture execution configuration (i.e., a $\langle core, frequency\rangle$ tuple) whenever an event is triggered and the corresponding event handler is executed such that it "barely" meets the performance target with minimal energy consumption. It is important to emphasize that one event may lead to multiple frames being updated. Therefore, the EBS runtime operates on a per-frame basis as frames are what ultimately dictate user perceivable experience. If an event execution only produces one frame, the runtime finds the ideal execution configuration for the single frame associated with the event. If an event's execution leads to a se-

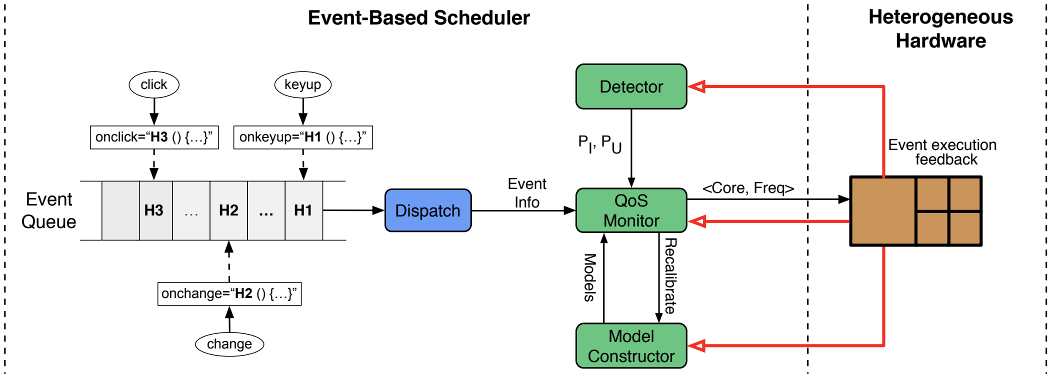


Fig. 5.10: Event-based runtime scheduling framework.

quence of frames such as in an animation, the runtime continuously identifies the ideal execution configuration for each frame until all the frames associated with the event are produced. All the associated frames share the same QoS target of the event.

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 Figure 5.10. 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-guided QoS hints such as those enabled by the `GreenWeb` language extensions as I will discuss in Chapter 6, 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 [164]. We will explain the detailed operation of the monitor in the next subsection. 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.

5.5.3 Scheduler Implementation Details

Performance Model We construct performance models for big and little cores separately. Each model predicts the event handler execution latency under different frequencies. We use the classical DVFS analytical model initially proposed in [178], and employed in subsequent work, such as [177]:

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

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 with the event latency under two different frequencies by calculating the value of T_{memory} and $N_{dependent}$. The trade-off in choosing the two frequencies is that on one hand using two sufficiently different frequencies provides higher accuracy, since the execution latencies from closer frequencies are more susceptible to measurement noise. But on the other hand, using two frequencies that are extremely high and low may result in execution falling in the imperceptible or unusable QoS regions, ultimately wasting energy. In our current implementation, we use the highest and the second-highest frequencies to construct the performance model. We find that the run-to-run variation for the data collected using these two frequencies is low, resulting in a robust model.

Frame Latency Tracking Tracking frame latency is crucial to con-

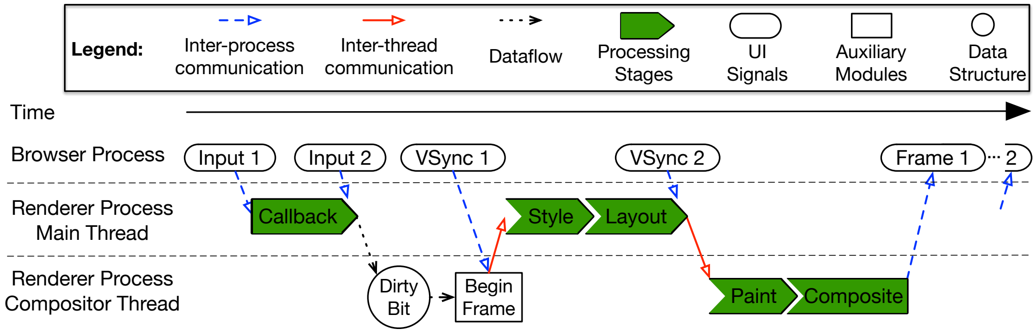


Fig. 5.11: The simplified view of frame lifetime in modern multiprocess/thread browsers. A frame starts when the browser process receives an input event and ends when the frame is displayed and the browser process is signaled. In between, an input event is processed by different stages spread across multiple threads. Different input events might interleave with each other.

structing the performance and energy model. However, accurate frame latency tracking is a nontrivial task, primarily because of the complexities involved in generating a frame in modern Web browsers. Most prior work either is concerned only with the callback latency [76, 183], which, as we will show later, contributes to only a portion of frame latency, or it considers logical latency (e.g., the number of conditionals evaluated), which is insufficient to construct the prediction models [151].

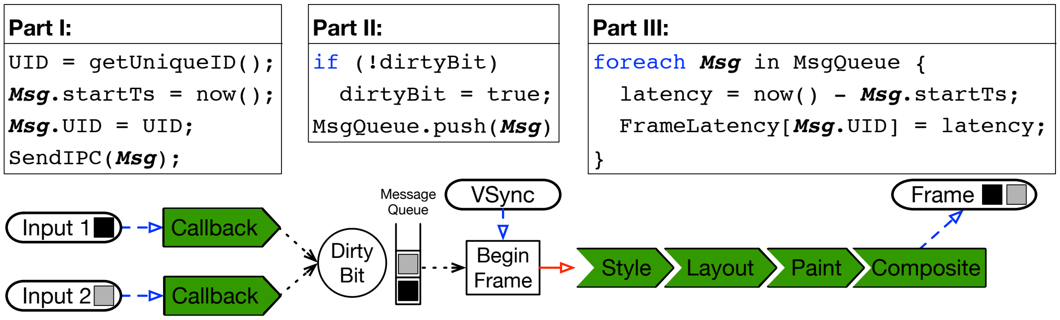
Accurately tracking frame latency requires us to understand how a frame is processed internally by a Web browser. Using Google Chrome browser as an example, Figure 5.11 illustrates a typical frame lifetime, starting from when an input event is received by the browser to when the frame is generated. Although we focus on Chrome, the execution model is generally applicable to almost all modern Web browsers such as Firefox, Safari, Opera, and Internet

Explorer.

The browser process receives an input event and sends it to the renderer process, which applies five processing stages to produce a frame: callback execution, style resolution, layout, paint, and composite [126]. In the end, the browser process receives a signal indicating that the frame is produced. To improve performance, the processing stages are spread across two threads, and some portion of the composite stage could be offloaded to GPU (not shown). Note that our performance model in Equation. **??** captures the GPU processing time.

The key to latency tracking is to accurately attribute a frame to its triggering input. Two complexities of the frame generation process make frame attribution non-trivial. First, different input events might be interleaved. For the example in Figure 5.11, Input 2 is triggered before Input 1 finishes. Naively associating an input event with its immediate next frame in this case would mistakenly attribute Frame 1 to Input 2.

Second, one frame might be associated with multiple input events. This is because modern browsers generate a new frame only when the display refreshes, i.e., a VSync signal arrives (typically 60 Hz on a mobile device), to avoid screen tearing [23, 38]. If multiple callback functions have been executed before a VSync arrives, their effects are batched and cause only one frame. The batching is achieved through a *dirty bit*. Each callback sets a dirty bit to indicate whether a new frame is needed as a result of callback execution. Callbacks from different inputs write to the same bit, but as long as one call-



Part I:

```
UID = getUniqueID();
Msg.startTs = now();
Msg.UID = UID;
SendIPC(Msg);
```

Part II:

```
if (!dirtyBit)
  dirtyBit = true;
MsgQueue.push(Msg)
```

Part III:

```
foreach Msg in MsgQueue {
  latency = now() - Msg.startTs;
  FrameLatency[Msg.UID] = latency;
}
```

Fig. 5.12: Frame tracking algorithm. The key idea is to attach each input event with a metadata (`Msg` in the code) that uniquely identifies an input event and is propagated with the event. We use two colors to represent metadata of two different events in this example.

back sets the dirty bit, a new frame will be generated when the browser later receives a VSync signal.

We show the flow of our tracking algorithm in Figure 5.12. The key idea is to attach each input event with a piece of metadata (`Msg` in the code) that is propagated with the event throughout the entire processing pipeline. Each `Msg` is assigned with an ID that uniquely identifies an input (Part I). To track batched input events, the dirty bit system is augmented with a message queue, which stores `Msg` metadata of all input events that access the dirty bit after the previous VSync. All messages in the queue get propagated when the VSync signal arrives (Part II). When the browser receives the frame ready signal, it iterates through all the messages propagated with the signal and calculates the frame latency of each input based on their unique ID (Part III).

Energy Model The energy model predicts the energy consumption of an event handler's execution. We construct the energy model on the basis of 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 browser upon every launch. Persistently storing and looking up the power profile file aligns with the Android standard [30]. Alternatively, we can dynamically derive the power consumption if power proxy counters, such as Running Application Power Limit (RAPL) [83], are available and exposed to software. In our case, a rough estimate of the power consumption is sufficient.

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

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

Overheads The QoS monitor accounts for scheduling overheads, which consist of two components: the overhead of the scheduling algorithm itself and the overhead of changing the architecture configuration (i.e, big/little core migration and/or frequency scaling). The scheduling algorithm's overhead is dominated by model construction, which only requires solving a two-variable linear system that imposes almost negligible overhead. For changing the architecture's configuration, we assume 100 μs for frequency scaling and 20 μs for switching cores, as discussed in Chapter 5.1.

5.5.4 Experimental Setup

Application Selection Table 5.2 shows the applications we use for evaluation. We crawl them using HTTrack [21] and host them on our Web server to enable annotations (discussed later). We acknowledge that the network condition could be slightly better when accessing a local server. However, we believe it has minimal impact because many prior work has shown that computation dominates the performance and energy consumption for today's mobile Web applications [112, 184, 185]. Overall, these applications cover a wide range of domains such as news, utility, etc., and are mostly among the top 200 websites as ranked by Alexa [55].

Baseline We compare EBS with two baselines:

- *Perf* is the policy that always runs the system at the peak performance, i.e., highest frequency in the big core in our setup. It is the standard policy for interactive applications to guarantee the best user QoS experience.
- *Interactive* is Android's default `interactive` CPU governor designed specifically for interactive usages. It maximizes performance when the CPU recovers from the idle state, and then dynamically changes CPU performance as CPU utilization varies [3].

Usage Scenarios Real-world user study over one year span from the LiveLab project [159] shows that mobile users often have to interact with devices under different battery conditions. Therefore, we evaluate EBS under two primary usage scenarios based on battery status:

- "Imperceptible" represents scenarios in which the battery budget is abundant and users expect high QoS experience. Therefore, the target performance that `WebRT` must deliver is high. We will rigorously define "imperceptibility" in Chapter 6.1.

- "Usable" represents scenarios in which the battery budget is tight and users could tolerate lower performance. Therefore, the target performance that `WebRT` must deliver is lower than the "imperceptible" scenario. We will rigorously define "usable" in Chapter 6.1.

It is worth noting that *Perf* and *Interactive* behave the same independently of the usage scenario. EBS under these two scenarios is denoted by EBS-I and EBS-U, respectively, in the rest of the evaluation.

5.5.5 Evaluation

In this section, we perform a sequence of interactions on each application, and evaluate the end-to-end behavior of EBS. Each sequence consists of a mix of LTM interactions and contains events with different QoS types and QoS targets. The "Full Interaction" category in Table 6.3 shows the details of each interaction. On average, each interaction sequence triggers about 94 events and lasts about 43 s.

We acknowledge that there are alternative ways to interact with each application. Thoroughly evaluating all the representative interactions with each application involves a large user study and is beyond the scope of this paper. However, we did perform our due diligence to make sure that the chosen interaction for each application is representative.

Representative Study We first use `Paper.js` as a case-study to illustrate the effectiveness of the runtime system. Figure 5.13 shows three rep-

Table 5.2: List of evaluated applications. “Interaction Description” provides a high-level description of the kind of interactions that are performed on each application. “Time” indicates the total interaction duration. “Events” indicates the amount of events triggered during an interaction.

Application	**Interaction Description**	**Time**	**Events**
`BBC`	Load the main webpage	0:86	60
`Google`	Load the main webpage	0:31	26
`CamanJS`	Tap a button to apply an image filter	0:49	24
`LZMA-JS`	Tap a button to compress a file	0:53	39
`MSN`	Tap to display the menu bar	0:59	126
`Todo`	Tap to delete all List items.	0:26	26
`Amazon`	Horizontally swipe the Ads bar	0:36	101
`Craigslist`	Scroll to find the “outdoor” category	0:25	22
`Paper.js`	Move finger to draw a series of curves	0:16	560
`Cnet`	Tap a button to expand the main menu	0:46	60
`Goo.ne.jp`	Tap button to switch to another news	0:16	23
`W3Schools`	Tap to show the sitemap	0:64	59

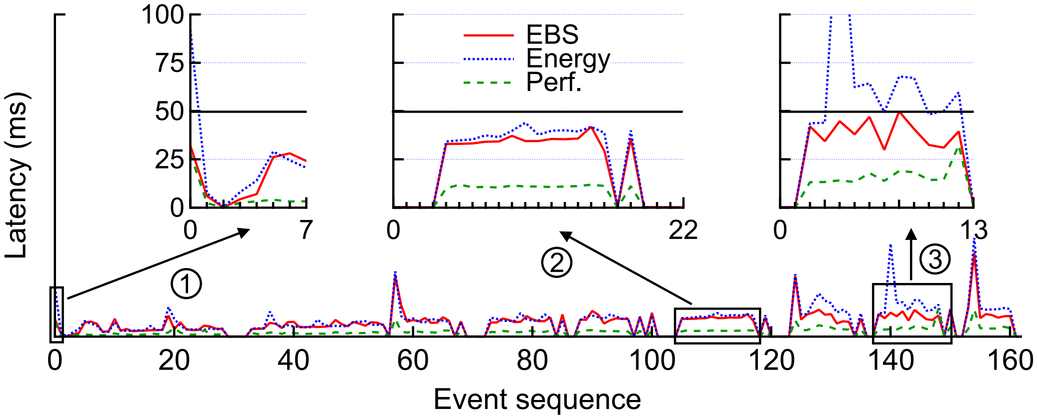
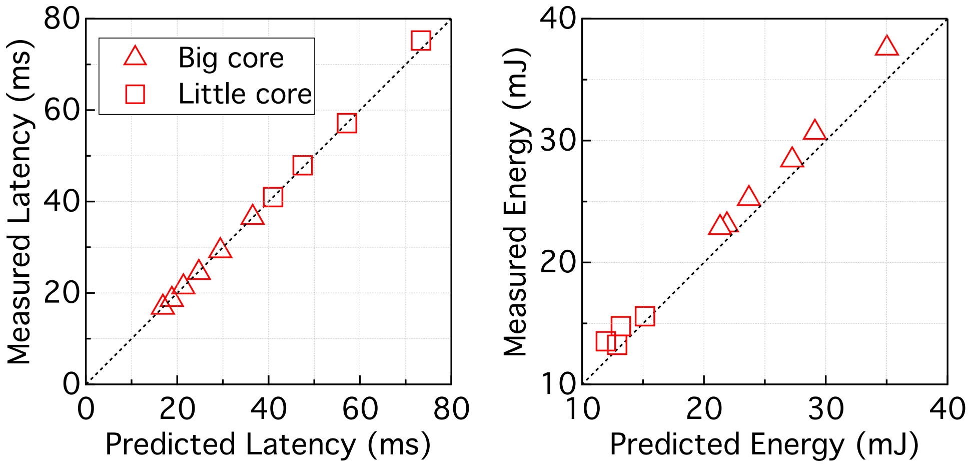


Fig. 5.13: The event execution trace of Paper.js under three different runtime schemesr.

resentative snapshots of `Paper.js`. The x-axis shows the event sequence during the execution because our runtime system works at the event granularity, and the y-axis shows the QoS under three different runtime management schemes: one always optimizes for the lowest energy (Energy), one always optimizes for highest performance (Perf), and the event-based scheduler (EBS). In this example, we set the QoS target to 50 ms. Intuitively, as the actual QoS nears the QoS target (without violating it), less energy is consumed.

We take three snapshots throughout application execution to discuss in more detail. The first snapshot shows the initial calibration period, which constructs the predictive QoS model. Figure 5.14 shows the accuracy of the model constructed for `Paper.js` by comparing the measured event latency to the predicted latency for each frequency. As shown, our simplified achieves an average error <0.3%. As a result, the runtime system in the calibration mode



(a) Latency model validation.

(b) Energy model validation.

Fig. 5.14: The latency and energy model accuracy for `Paper.js`, which has the median accuracy of all the applications.

settles down quickly after the fifth run. The near-perfect accuracy and low performance overhead justifies our model simplification decision.

After calibration, the runtime enters monitoring mode. The second snapshot in Figure 5.13 illustrates typical operation for monitoring mode when events are so lightweight they could solely meet QoS on the most energy-optimized performance configurations. However, such a scheme is infeasible because event behavior is not known a priori. Our runtime system is able to quickly adapt to the constant influx of simple events to achieve behavior similar to energy-oriented optimization scheme.

The third snapshot illustrates an application phase with more complex events where the energy-oriented scheme is rarely able to meet the QoS target. Although the performance scheme maximizes the likelihood to meet the QoS

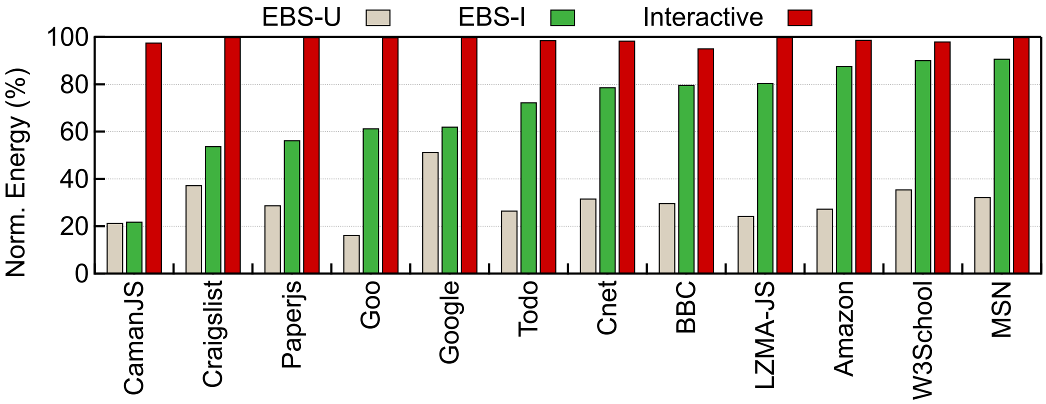
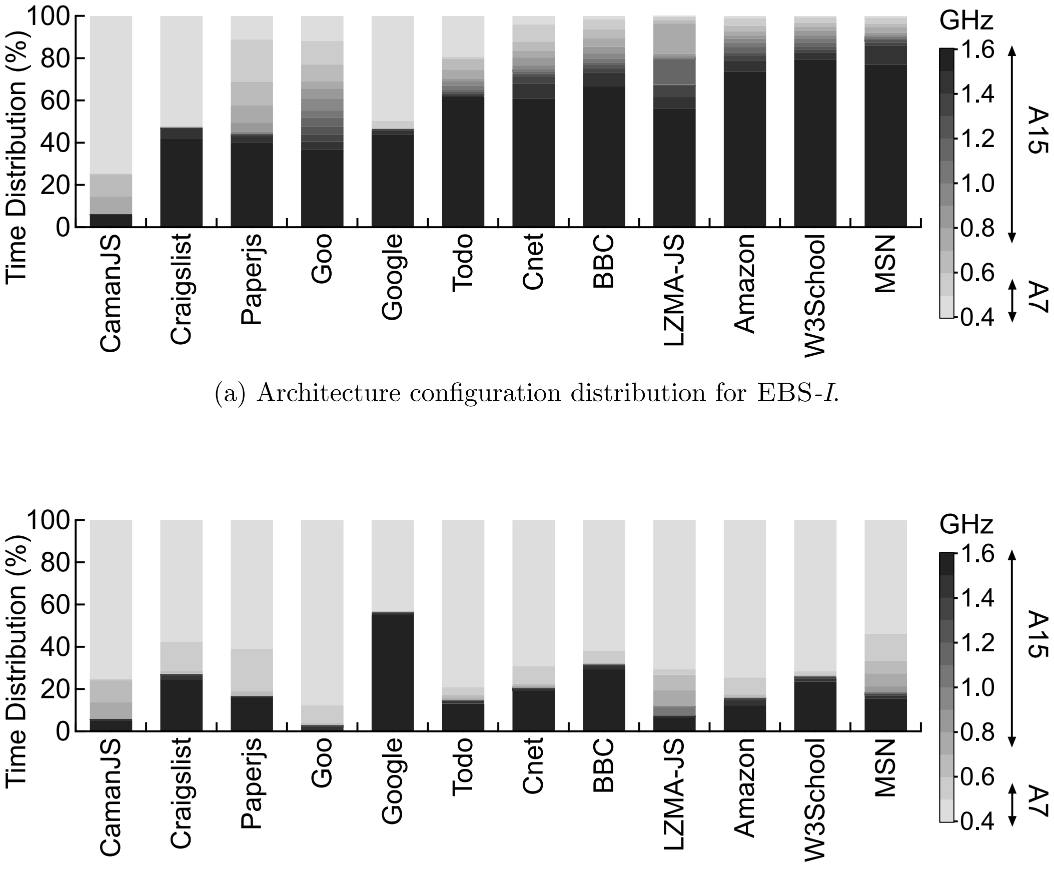


Fig. 5.15: Energy consumption normalized to *Perf*. Lower is better.

requirement, it results in excessively high energy consumption. Our eQoS runtime intelligently provisions architecture resources to prolong execution within the target QoS bounds. Exploiting the QoS gap between peak QoS and target QoS results in a significant energy savings.

Energy Savings We now expand our evaluation to all benchmarked applications. Figure 5.15 shows the energy consumption of *Interactive* and EBS's two usage scenarios. The results are normalized to *Perf*, and sorted in the ascending order of EBS-*I*. As compared to *Interactive*, EBS achieves on average 29.2% and 66.0% energy saving under the imperceptible and usable usage scenarios, respectively.

Interactive consumes energy close to *Perf* across all applications, indicating that the Android `Interactive` governor is almost always operating at the peak performance. This is because user interactions, especially events with a "continuous" QoS type, typically generate a large volume of frames, which



(a) Architecture configuration distribution for EBS-I.

(b) Architecture configuration distribution for EBS-U.

Fig. 5.16: The architecture configuration distribution under the "imperceptible" (EBS-I) and "usable" (EBS-U) usage scenario.

leads to high CPU utilization. *Interactive* responds to the high CPU utilization by increasing CPU performance. With the QoS knowledge provided by developers, however, EBS can identify execution configurations that conserve energy while still meeting QoS requirements.

Architecture Configuration Distribution To better understand the sources of energy savings of EBS, we examine the architecture configuration distribution of EBS under the imperceptible and usable usage scenario shown in Figure 5.16a and Figure 5.16b, respectively. Bars with darker colors indicate higher performance configurations.

We make two notable observations from the distribution results. First, EBS tends to bias toward big core (A15) configurations much more often under the imperceptible scenario (Figure 5.16a) than under the usable scenario (Figure 5.16b). This observation confirms the result that EBS-*I* has less energy saving than EBS-*U*. Second, the fact the EBS dynamically changes its execution configuration under different QoS targets indicates that the EBS can adapt to different user QoS expectations while saving energy. In contrast, *Interactive* always adopts the same scheduling policy independent of the user QoS expectation, leading to energy waste. This observation indicates that an ACMP architecture is beneficial in mobile Web, but the burden is on the runtime system to intelligently leverage it.

Configuration Switching Frequency Complementary to the distribution of architecture configuration, Figure 5.17 shows the switching frequency of architecture configuration in EBS-*I* and EBS-*U*. We decompose

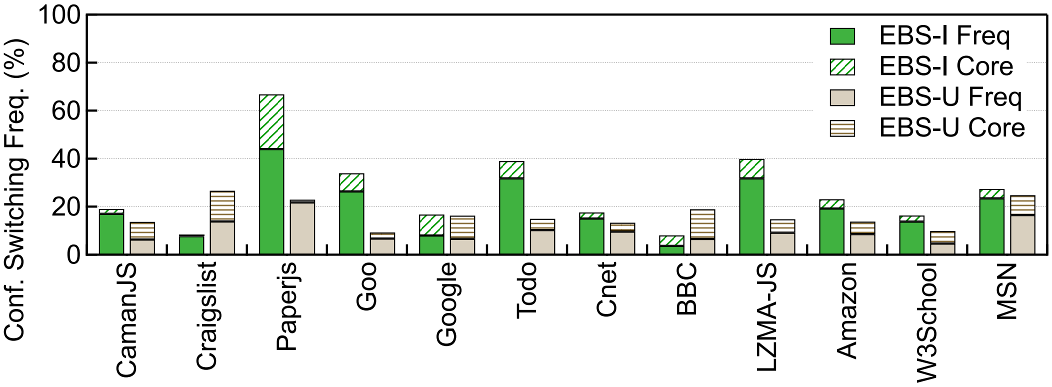
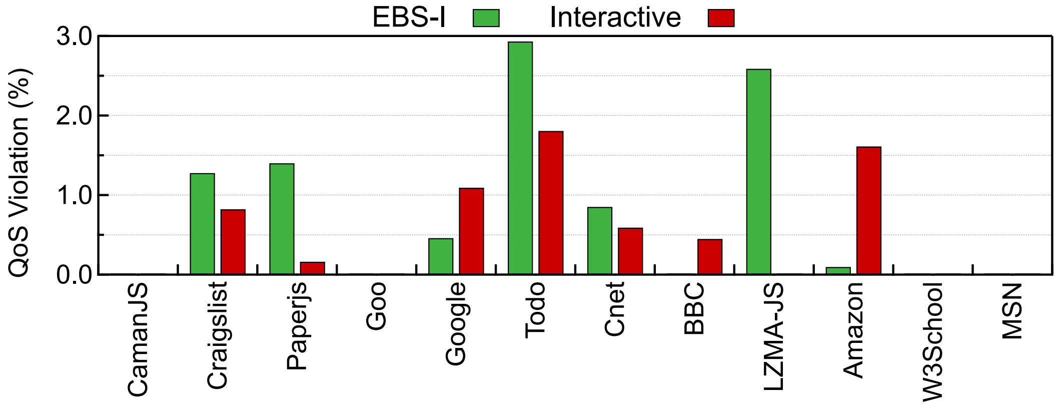


Fig. 5.17: Execution configuration switching frequency under EBS-*I* and EBS-*U*. Two configuration switching types exist: CPU frequency switch (solid) and core migrations (stripe).

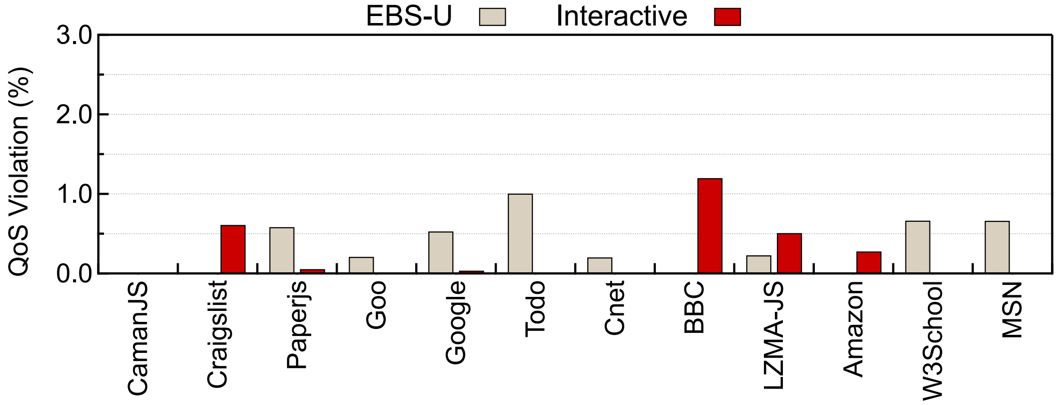
the configuration switching into two categories: CPU frequency change and core migration (between big and little clusters). Thus, Figure 5.17 is shown as a stacked bar plot where the frequencies of both categories are stacked for each application.

We draw three conclusions from the switching frequency statistics. First, EBS introduces only modest configuration switching (20% on average). Recall from Chapter 5.1 that the CPU frequency switching and core migration incur overhead only to the order of μs, much smaller than the QoS target which is typically to the order of millisecond. Therefore, the execution configuration has minimal performance impact.

Second, for most of applications EBS-*I* incurs more switchings than EBS-*U*. This is unsurprising because as compared to EBS-*U*, EBS-*I* optimizes for a tighter QoS target, which is more sensitive to frame (phase)



(a) QoS violation comparison under the imperceptible usage scenario.



(b) QoS violation comparison under the usable usage scenario.

Fig. 5.18: QoS violations are presented as additional violations on top of *Perf*. The y-axis of the two figures are kept the same for comparion purposes.

variance and more vulnerable to frame performance mis-prediction. In contrast, a more relaxed QoS target is more robust against frame variance. Our results suggest that a better frame performance predictor such as the profiling-guided prediction [133] would be helpful in reducing the execution configuration switching in the imperceptible mode.

Third, the CPU frequency change dwarfs core migrations and dominates the configuration switching. Thus, fast DVFS is desired. Our results suggest that a fast on-chip voltage regulator that is increasingly prevalent in server processors [68, 117] is also beneficial in mobile CPUs.

QoS Violation Figure 5.18a and Figure 5.18b show the QoS violation of *Interactive* and EBS under the imperceptible and usable scenarios, respectively. On average, EBS introduces 0.8% and 0.6% more QoS violations than *Perf* under the imperceptible and usable scenarios, respectively. The QoS violations are lower than in the microbenchmarks because the interaction duration gets longer and the QoS violations caused by profiling runs are amortized.

Compared to *Interactive*, EBS has similar, in some cases fewer, QoS violations. Considering the significant energy savings, we conclude that the QoS-aware EBS system can use energy more wisely by being aware of user QoS expectations. Overall, EBS achieves better energy efficiency than the QoS-agnostic *Interactive* scheme.

5.6 Related Work

We first discuss prior scheduling work on ACMP because the particular implementation of WebRT presented in this work relies on the ACMP architecture (Chapter 5.6.1). As the goal of WebRT is to save energy without sacrificing performance, we then discuss prior work on improving the performance (Chapter 5.6.2) and energy consumption (Chapter 5.6.3).

5.6.1 Single ISA/DVFS Scheduling

The particular implementation of WebRT is an example of utilizing single-ISA heterogeneous systems capable of DVFS for trading off performance with energy [124]. Nvidia's Kal-El [45] is a single-ISA heterogeneous system that integrates four high-frequency cores with one low-frequency core. ARM's proposed big.LITTLE system [43] 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 such as Samsung and Qualcomm [20]. We expect our WebRT implementation to be readily applicable to commodity mobile hardware.

The scheduling mechanism in WebRT differs from existing single-ISA scheduling and DVFS techniques in three key aspects: scheduling *unit*, scheduling *objective*, and scheduling *heuristics*. First, the scheduling unit in existing techniques is either interval based (fixed-instruction interval [124, 134, 139, 146, 154] or fixed-time interval [82, 98, 147, 170, 174]) or a code segment (e.g., critical sections, lagging threads, application kernels [71, 114, 115, 166]).

The scheduling unit in the WebRT is Web application-specific entities: event handlers in event-based scheduling and webpage loadings in webpage-aware scheduling. These Web application-specific units directly correspond to user interactions and let us directly optimize for user QoS experience.

Second, the scheduling objectives in existing techniques are typically architecture-level energy-efficiency metrics such as energy, EDP [95], and million-instructions-per-joule (MIPJ) [174]. These metrics trade off raw performance instead of QoS with energy. Therefore, they may lead to executions that fall into the imperceptible or unusable QoS regions and waste energy. On the contrary, WebRT scheduler aims at minimizing energy consumption under explicit performance constraint in order to guarantee satisfactory QoS experience.

Third, the webpage-aware scheduler's prediction-based scheduling heuristics is similar to other recent heterogeneous scheduling proposals in the architecture community [82, 134, 146, 154]. In contrast, instead of relying on (micro)architecture- and system-level statistics for prediction, the webpage-aware scheduler captures the complex behavior of webpage characteristics using regression modeling, and accurately predicts the webpage load time and energy consumption.

Timer coalescing [46] used in OS X Mavericks also exploits the performance slack for energy savings, similar to our event-based scheduling. It postpones noncritical timers and coalesces them for batch executions to increase the processor idle time for energy savings. However, timer coalescing applies only to timers in Apple's native applications (or OS processes), whereas our EBS framework is not limited to timer events, but can apply to any event-driven applications.

5.6.2 Web Performance Optimizations

Most prior research focus on parallelizing browser tasks, such as parsing, CSS selection, etc. [59, 130, 137, 138]. 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.

Another portion of performance optimizations focuses on improving the execution model of the Web browser through asynchronous/multiprocess rendering, resource prefetching, smarter browser caching, etc. [29, 41, 135, 136, 181]. All these techniques are orthogonal and can be integrated with my proposal, which primarily focused on the core rendering engine of a Web browser.

5.6.3 Web Energy Optimizations

Thiagarajan et al. [168] 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. Other works analyze the power/energy consumption of the entire smartphone[67, 74, 150], whereas we focus on improving the energy-efficiency of the mobile processor in response to the demand for high-performance.

Another body of energy-related research focuses on diagnosing energy bugs and hogs in mobile applications. These techniques either completely kill an energy-hungry application [144] or require developers to improve manually the energy efficiency [106, 132, 150]. `WebRT` eases developers' effort by automatically optimizing for energy efficiency.

Chapter 6

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, as mobile users become increasingly aware of poor energy behavior of applications [2, 35], Web developers today must be explicitly conscious of energy efficiency. Current programming language abstractions, however, provide developers few opportunities to optimize for energy efficiency. Instead, energy optimizations are mostly conducted at the hardware and OS level via techniques such as dynamic voltage and frequency scaling. Although effective from a system perspective, the key limitation of these techniques is that they are not aware of user quality-of-service (QoS) expectations and may lead to poor experience [133, 183, 185]. Failing to deliver a desirable QoS experience can cause severe consequences. For example, a 1-second delay in webpage load time costs Amazon $1.6 billion annual sales lost [87].

In this chapter, I present `GreenWeb`, a set of Web language extensions defined as Cascading Style Sheet (CSS) rules 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.

To help Web developers reason about QoS constraints in Web applications, our *key insight* is that user QoS experience can be sufficiently captured by two fundamental abstractions: *QoS type* and *QoS target*. Intuitively, QoS type characterizes whether users perceive QoS experience by interaction responsiveness or animation smoothness, and QoS target denotes the performance level that is required to deliver a desirable user experience for a specific QoS type. `GreenWeb` provides specific language constructs for expressing the two QoS abstractions and thus empowering Web developers to provide "hints" to guide energy optimizations.

Allowing programmers to annotate QoS information in applications is 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.

The rest of this chapter is organized as follows. Chapter 6.1 discusses the relationship between QoS, performance, and energy consumption. It lays the foundation of abstracting user QoS experience. Chapter 6.2 defines two abstractions that are critical to mobile user QoS experience, and Chapter 6.3 describes the proposed GreenWeb language constructs that express the two abstractions. Chapter 6.4 presents AutoGreen to demonstrate the feasibility of automatically applying GreenWeb annotations to a Web application. Chapter 6.5 discusses the relationship between the GreenWeb extensions and the WebRT runtime, and argue that GreenWeb and WebRT are synergistic while independent. Chapter 6.7 discusses the implications and limitations of the current design and implementation of GreenWeb. Finally, Chapter 6.8 puts GreenWeb in the general context of language support for performance and energy-efficiency.

6.1 Trade-off Between QoS, Performance, and Energy

We illustrate the relationship between application QoS, performance, and energy savings in Figure 6.1. 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 [72, 89, 140–142, 160] indicates that interactive application QoS can be classified into three distinct states as machine performance degrades: *imperceptible* $[P_H, P_I]$,

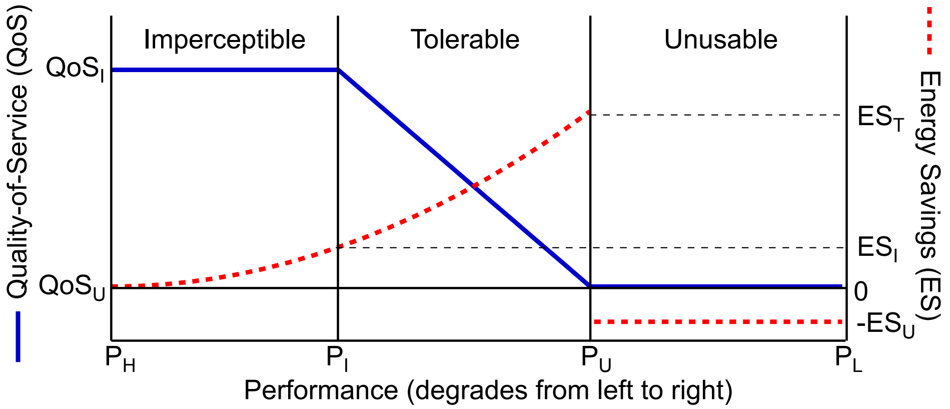


Fig. 6.1: The interplay between QoS, performance, and energy.

tolerable $(P_I, P_U]$, and *unusable* $(P_U, P_L]$.

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 [179, 182]. 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.

6.2 QoS Abstractions for Web Applications

Expressing user QoS experience to the underlying system is the key in QoS-aware energy efficiency optimizations. However, today's Web languages do not allow expressing QoS information. Programmers need new *abstractions*. We propose two abstractions, QoS type and QoS target, that capture two fundamental aspects of user QoS experience in Web applications. Such QoS abstractions hide the complexity of the specific application implementation from underlying systems while still providing enough details to guide energy optimizations. This section introduces the abstractions, and the next section (Chapter 6.3) describes the proposed language constructs that enable programmers to express the abstractions.

Abstracting user QoS experience requires us to first understand how users assess QoS experience in mobile Web applications. To that end, we leverage the LTM user-application interaction model described in Chapter **??**. The LTM model captures three fundamental user interactions in mobile Web applications (Loading-Tapping-Moving) and gives us a framework for reasoning about user QoS experience. Based on the LTM model, we propose the QoS type (Chapter 6.2.1) and QoS target (Chapter 6.2.2) abstractions. We discuss why they are necessary and sufficient to express QoS information for QoS-aware energy efficiency optimizations.

6.2.1 QoS Type

We define an abstraction called *QoS type* to capture different ways that users interpret the 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. Let us use the LTM model to elaborate on the two QoS types.

Single 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 [89]. For instance, imagine a fingertap interaction (T) 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 (L) also

Table 6.1: Interactions in mobile Web applications fall into three categories based on different QoS type and QoS target combinations.

QoS Type	**QoS Target** (P_I, P_U)	**Description**	**Interaction**
Continuous	(16.6, 33.3) ms	QoS experience is evaluated by *continuous* frame latencies.	T, M
Single	(100, 300) ms	QoS experience is evaluated by *single* frame latency. Users expect *short* response period.	T
	(1, 10) s	QoS experience is evaluated by *single* frame latency. Users expect *long* response period.	L, T

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 of the "first meaningful frame" [156], which indicates that a Web application is usable by users.

Under the "single" QoS type, an ideal energy-efficient system would allocate just enough energy to produce the single response frame and conserve energy afterwards. It is worth noting that the system might not be completely idle after the response frame is delivered. The system could still perform work such as updating the browser cache, performing garbage collection, or rasterizing off-screen pixels. Such "post-frame" work is not critical to user QoS experience and could be executed in a low-power mode.

Continuous 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 specific frame as in the "single" case. Ideally, an energy-efficient Web runtime would allocate just enough energy for each frame in the sequence and conserve energy after all the frames are produced.

Continuous frames are often found in the form of animations. The simplest form of animation is triggered by finger moving (M) such as scrolling. Tapping (T) 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 [37] and CSS animation/transition [11, 14].

Distinguishing between "continuous" and "single" is important. If an event callback triggers an animation but the runtime treats its QoS type as "single", the runtime would optimize for only the first frame in the sequence, and thus mis-operates for the remaining frames. On the other hand, if an event produces only a single frame followed by some "post-frame" work, a runtime (mistakenly) optimizing for "continuous" frame latency would force the hardware to run at the peak performance to execute the "post-frame" work (with the intention of generating more frames), leading to energy waste. Whether an event triggers a single frame or a sequence of frames can not be determined a priori. In Chapter 6 we will introduce a set of language extensions that let developers explicit specify an event's QoS type, through which the runtime could be better informed in optimizations.

6.2.2 QoS Target

Another critical QoS abstraction is *QoS target*, denoting the performance level needed to deliver a certain QoS experience. 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.

Two different QoS targets exist that are critical to user experience: imperceptible target (P_I) and usable target (P_U) [183]. Imperceptible target delivers a latency that is imperceptible/instantaneous to users. Achieving a performance higher than P_I does not add user perceptible value while unnecessarily wasting energy. The usable target, in contrast, corresponds to a latency that can barely keep users engaged. Delivering a performance lower than P_U may cause users to deem an application unusable and even abandon it.

Single For interactions with the "single" QoS type, QoS target depends on the complexity of the interaction [89]. 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 [33]. 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 [142]. Psychological study shows 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 [72, 140]. Therefore, 1 second and 10 seconds can be treated as the P_I and P_U values for "heavyweight" interactions, respectively.

Continuous For interactions with a "continuous" QoS type, 60 and 30 frames per second (FPS) deliver a "seamless" and "just playable" user experience, respectively [80]. 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 [23, 26].

User interactions fall into three distinct categories based on the different QoS type and QoS target combinations as listed in Table 6.1. 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.

6.3 QoS-Aware Web API Design

We now present `GreenWeb`, a set of Web language extensions that lets application developers easily express the two QoS abstractions as program annotations. We first discuss the design principles of `GreenWeb` extensions (Chapter 6.3.1). We then describe the design and specification of `GreenWeb` (Chapter 6.3.2). We then present usage scenarios to demonstrate the expressiveness and modularity of the `GreenWeb` design (Chapter 6.3.3).

6.3.1 Design Principles

`GreenWeb` extensions follow three design principles. First, adding or removing `GreenWeb` annotations does not change application functionality and correctness. In other words, `GreenWeb` annotations are *modular* components in an application. Modularity allows developers who are unfamiliar with application logic to still be able to express QoS information, and allows removing problematic annotations in a non-interference manner.

Second, `GreenWeb` is *intuitive* for Web application developers to use in that it does not require developers to specify absolute values of QoS targets (although this option is provided if needed). This is important because developers may not have quantitative knowledge about user perceptibility. Instead, developers provide a qualitative specification. This design makes the APIs more expressive, and provides flexibility for runtime implementation.

Third, `GreenWeb` syntax is *compatible* with current Web language specifications, which is crucial for lowering the learning curve and ensuring pro-

```
GreenWebRule ::= Selector? { QoSDecl+ }
    Selector ::= Element:QoS
     QoSDecl ::= CDecl | SDecl
       CDecl ::= onEventName-qos: continuous[, v, v]
       SDecl ::= onEventName-qos: SValue
      SValue ::= single, short|long|[v, v]
```

Element	DOM element	*v*	Integer value
EventName	DOM event name		

Fig. 6.2: The syntax of `GreenWeb` language extensions.

grammer productivity.

6.3.2 QoS-Aware Web API Design

The `GreenWeb` APIs extend the 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*. 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`.

Traditionally, CSS supports purely visual style properties such as fonts and colors. Recent development of CSS (e.g., CSS3) lets developers express

Table 6.2: Specifications of the `GreenWeb` APIs. Each API is a new CSS rule specifying the QoS information when a particular event is triggered on certain Web application 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 6.1 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 (long) latency. Use the P_I and P_U values in Table 6.1 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.

richer information such as controlling animations [11] and adapting to different device form factors [16]. GreenWeb follows this spirit of CSS language evolution and further expands the CSS semantics scope to allow expressing user QoS related information.

Figure 6.2 shows the GreenWeb syntax, and Table 6.2 lists the semantics of each 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 the GreenWeb extensions.

Selector To decorate a CSS rule as specifying the QoS information of an element, we define a new CSS pseudo-class selector [12] ":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 6.2). The semantics of declaring onevent-qos: continuous is that as soon as onevent is triggered on element E, the Web browser 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 6.2). 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; the `GreenWeb` runtime will use the P_I and P_U values in Table 6.1 as the default QoS target. However, we also provide the flexibility for developers to overwrite the default QoS targets. This is achieved by specifying absolute values of P_I and P_U (in milliseconds) after `single` or `continuous`, as shown in the third rule in Table 6.2.

6.3.3 Example Usage

The proposed QoS-aware `GreenWeb` APIs support a wide range of Web application interaction patterns. We explore different usages using two examples.

Animations via CSS Transition The first example involves annotating events that achieve animation using a CSS transition. A CSS transition lets developers specify the initial and end state of an animation and how long the transition takes, while leaving the transition implementation to the Web browser [14]. Figure 6.3 shows an example in which the transition of the `width` property of a `div` element is animated. The initial `width` property is set to

```
 1 <html> <head>
 2   <style>
 3     div#example {
 4       width: 100px;
 5       transition: width 2s;
 6     }
 7     div#ex:QoS {
 8       ontouchstart-qos: continuous;
 9     }
10   </style>
11   <script>
12     function animateExpand() {
13       var node = document.getElementById(‘ex’)
14       node.style.width="500px";
15     }
16   </script> </head>
17
18   <body>
19     <div id=“ex” ontouchstart=“animateExpand()”>
20     </div>
21     <!-- many elements -->
22 </body> </html>
```

Fig. 6.3: Express the QoS type of `ontouchstart` event as "continuous," and use the default P_I and P_U values.

100px (line 4). The style declaration "`transition: width 2s;`" at line 5 indicates that whenever the `width` property is reset, the transition will begin and finish in 2 seconds. Later when users click the `<div>` element, the `animateExpanding` callback is executed (line 19), which sets the `width` property to 500px, triggering the 2-second animation.

Application developers realize that user QoS experience of the `ontouchstart` event is dictated by the animation smoothness. Using `GreenWeb`, developers

```
 1 <html> <head>
 2   <style>
 3     body:QoS {
 4       ontouchmove-qos: continuous, 20, 100;
 5     }
 6   </style>
 7   <script>
 8     var latestY = 0, ticking = false;
 9     function animateMove() {
10       latestY = window.scrollY;
11       if(!ticking)
12         requestAnimationFrame(function() {
13           ticking = false;
14           /* Animation code omitted */
15         });
16       ticking = true;
17     }
18   </script> </head>
19
20   <body ontouchmove=“animateMove()”>
21     <!-- many elements -->
22 </body> </html>
```

Fig. 6.4: Express the QoS type of `ontouchmove` event as “continuous,” and use 20 ms and 100 ms as the new QoS targets.

could express such information by specifying that the QoS type of `ontouchstart` event is “continuous” (lines 7-9). Without further expressing the QoS targets, the default values of T_I and T_U (16.6 ms and 33.3 ms) are used.

Animations via rAF Another common way of achieving animation is through the `requestAnimationFrame` (rAF) functions. Figure 6.4 shows the code snippet. In a nutshell, every time a user moves a finger, `rAF` is executed (if not already) to register an anonymous callback function (line 12), which will get executed when the display refreshes (i.e., when a VSync signal [38] arrives) [23] to achieve animation.

Application developers realize that once move events start, they trigger a sequence of continuous frames that determine user QoS experience. In addition, the developers believe that the specific animation in this application does not require a high FPS. Therefore, they specify the QoS type as "continuous" and overwrite the default QoS targets with 20 ms and 100 ms, respectively (lines 3-5).

Modular Design Discussion The `GreenWeb` API design is modular in the sense that `GreenWeb`-annotated program can be integrated with other non-annotated code while ensuring functionality. After all, `GreenWeb` extensions concern with QoS and traditional language constructs concern with functionality. This composability ensures a *separation of concerns* between QoS and functionality (correctness) in Web programming.

The modularity of `GreenWeb` also lets developers add QoS annotations for an event independent of how the event callback is implemented. For example, although animations in the above two examples are implemented through different mechanisms (CSS transition and `rAF`) and are triggered by different events (`ontouchstart` and `ontouchmove`), developers simply express the QoS type as "continuous" without having to understand the implementation details. One can imagine that the modularity of the `GreenWeb` APIs would also allow annotating QoS information for functionalities that are implemented by thirty-party libraries whose source code is not available. Modularity is important for extending Web languages because Web application implementations change rapidly. The `GreenWeb` annotations can remain unchanged as the application version evolves and can be removed without interfering the rest of the application logic.

6.4 Automatic Annotation

To assist programmers in annotating a Web application with QoS information, we provide a system called AutoGreen, which automatically applies `GreenWeb` annotations. The rationale behind designing AutoGreen is twofold. First, some Web developers may not want to spend the extra effort of manual annotation, such as for legacy applications. Second, in complex Web applications with many DOM nodes and events, manually going through all events could be cumbersome. In both cases, AutoGreen automatically finds all the events and annotates them with the two QoS abstractions, enabling QoS-aware energy efficiency optimizations without programmer intervention.

Figure 6.5 gives an overview of AutoGreen. It consists of three major phases: an instrumentation phase, a profiling phase, and a generation phase. The instrumentation phase first discovers all DOM nodes and their associated events in an application, and instruments every event callback to inject QoS detection code. With the instrumented application, AutoGreen performs a profiling run of each event by explicitly triggering its callback function. During the callback execution, the (injected) detection code checks for certain conditions to determine an event's QoS type and QoS target. After profiling,

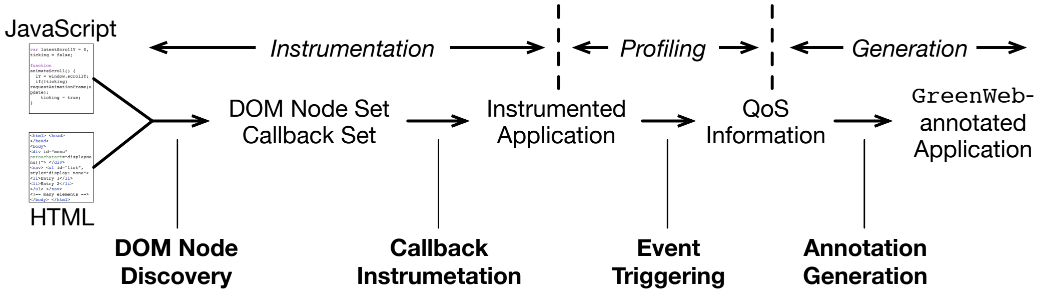


Fig. 6.5: AutoGreen's workflow to automatically annotate mobile Web applications with `GreenWeb` APIs.

AutoGreen generates QoS annotations and injects them back to the original code.

The detection code determines the QoS type of an event as follows. An event's QoS type is set to "continuous" if its callback function triggers a jQuery `animate()` function, a `rAF`, or a CSS transition/animation. Otherwise the QoS type is set to "single." To detect `animate()` and `rAF`, we overload their original functions and check in the overloaded function. To detect CSS transition/animation, we register a `transitionend/animationend` event [10, 13], which if triggered indicates that a CSS transition/animation exists. As a proof-of-concept prototype, our current implementation does not yet support checking other ways of realizing animations, but could be trivially extended to do so by following a similar detection procedure as described above.

AutoGreen uses the default QoS target values listed in Table 6.1 for each detected QoS type. Particularly, for events with a "single" QoS type, AutoGreen always assumes a short duration. This is because AutoGreen does not understand the semantics of an event callback function and has to make conservative decisions about the QoS target information in favor of satisfying QoS over conserving energy.

6.5 GreenWeb and WebRT Inteplay

GreenWeb and WebRT are independent. On one hand, GreenWeb does not make any specific assumptions about how the underlying runtime is implemented as long as the runtime is able to make trade-offs between performance and energy consumption. For example, 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 [48, 133]. In addition, one could implement a GreenWeb runtime using pure software-level techniques, such as prioritizing resource loading [69] or using power-conserving colors [85].

On the other hand, WebRT does not assume how the QoS requirements is provided to be able to perform scheduling. If not annotated using GreenWeb APIs, the QoS information could be defaulted to values that a particular implementation assumes. That is, the WebRT schedulers can assume a default QoS target and QoS type for each Loading, Touching, and Moving interaction. Moreover, the WebRT schedulers could also attempt to infer the QoS type and QoS target by profiling event executions to identify which category in Table 6.1 that an event falls into.

6.6 Evaluation

We evaluate GreenWeb in two different aspects. First, we evaluate the "effectiveness" of the GreenWeb APIs (Chapter 6.6.1). That is, given the annotated QoS information, can a Web runtime effectively optimizes for energy-efficiency while meeting the QoS expectations? Finally, we evaluate the annotation effort in apply GreenWeb APIs to Web application (Chapter 6.6.2). That is, is the annotation effort lightweight enough to incentive developers to use GreenWeb APIs?

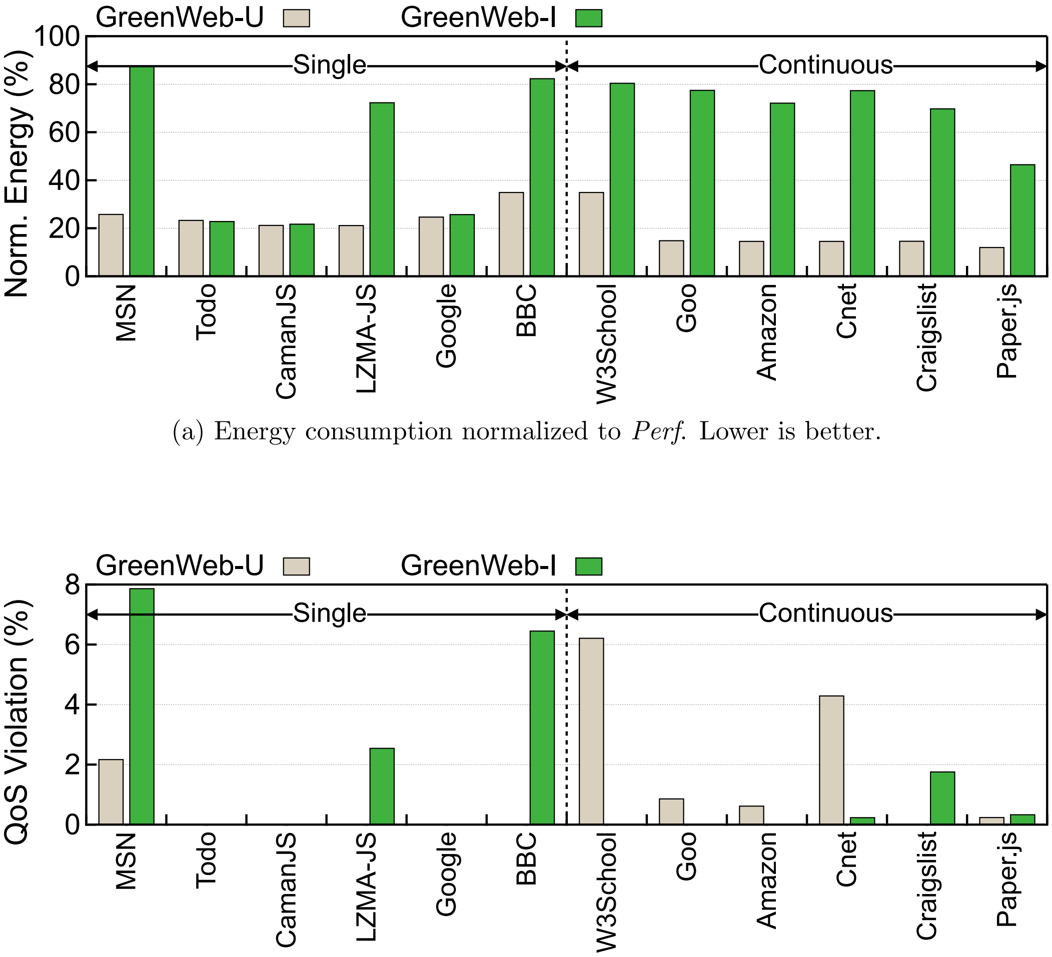
To evaluate GreenWeb we implement a WebRT-based GreenWeb runtime–although note that WebRT and GreenWeb are independent as discussed in Chapter 6.5. In particular, the WebRT implementation is exactly the same as what we described in Chapter 5, i.e., based on the ACMP architecture. The software and hardware platform we use to the evaluate GreenWeb is the same as described in Chapter 5.1. In addition, we base our evaluation on the same set of applications that are used in evaluating EBS in Chapter 5.5.5. The details of each application in shown in Table 6.3

6.6.1 Energy-Efficiency Improvement

To understand the effectiveness of GreenWeb on *individual* events, we design a set of microbenchmarking experiments. The goal is to exercise GreenWeb on various events that differ in interaction types (LTM), QoS type, and QoS target. To better understand the behavior of GreenWeb during microbenchmarking, we compare it only against *Perf*, which always has the highest energy

Table 6.3: Applications used for evaluating GreenWeb. They are the same as the ones used for evaluating EBS in Chapter 5.5.5. "Annotation" indicates percentage of events that are annotated. "Events" indicates the amount of events triggered during full interaction. Note: we only annotate and count events that are directly triggered by mobile user interactions as discussed in Chapter 5.2. Applications marked with * are manually annotated because they are developed using libraries that AutoGreen does not currently support. Their annotation percentage numbers are estimated.

Application	**Interaction**	**QoS Type**	**QoS Target**	**Events**	**Annotation**
BBC	Loading	Single	(1, 10) s	60	20%*
Google	Loading	Single	(1, 10) s	26	87.5%
CamanJS	Tapping	Single	(1, 10) s	24	100%
LZMA-JS	Tapping	Single	(1, 10) s	39	100%
MSN	Tapping	Single	(100, 300) ms	126	51.2%
Todo	Tapping	Single	(100, 300) ms	26	38.3%
Amazon	Moving	Continuous	(16.6, 33.3) ms	101	33%*
Craigslist	Moving	Continuous	(16.6, 33.3) ms	22	84.6%
Paper.js	Moving	Continuous	(16.6, 33.3) ms	560	100%
Cnet	Tapping	Continuous	(16.6, 33.3) ms	60	55.3%
Goo.ne.jp	Tapping	Continuous	(16.6, 33.3) ms	23	51.8%
W3Schools	Tapping	Continuous	(16.6, 33.3) ms	59	100%



(a) Energy consumption normalized to *Perf*. Lower is better.

(b) QoS violations normalized to *Perf*. Lower is better.

Fig. 6.6: Microbenchmarking results. Energy numbers are normalized to *Perf*, which provides the best QoS and consumes the most energy. QoS violations are presented as additional violations on top of *Perf*. *GreenWeb-I* and *GreenWeb-U* are `GreenWeb` under two usage scenarios.

and lowest QoS violation.

We construct the microbenchmark set by pairing each application in Table 6.3 with one of the three primitive interactions (Loading, Tapping, Moving). For each interaction, we manually apply `GreenWeb` annotations. The QoS type and QoS target are determined by the authors visually observing the effect of each interaction. The "Micro-benchmarking" category in Table 6.3 shows details about each microbenchmark. Half of the interactions have a QoS type of "single", and the other half have a QoS type of "continuous." Overall, our microbenchmarks cover user events that have different combinations of interaction types, QoS types and QoS targets.

Energy Savings Figure 6.6a shows the energy consumption of `GreenWeb` under both the imperceptible (*GreenWeb-I*) and the usable (*GreenWeb-U*) usage scenarios. The results are normalized to *Perf*. For the diverse set of interactions in our microbenchmark, `GreenWeb` achieves an average 31.9% and 78.0% energy saving under the two usage scenarios, respectively. Overall, the energy savings under the usable mode are higher than in the imperceptible mode because the usable QoS target is lower, which allows `GreenWeb` to leverage low energy ACMP configurations more often.

The greatest energy savings in the imperceptible mode come from events in the application Todo, CamanJS, and LZMA-JS. All three events have a "single" QoS type, but with different QoS targets (100 ms and 1 s). The frame complexity of the three events is low relative to their QoS target such that `GreenWeb` can meet the QoS target using only little core configurations. *Perf* wastes energy by constantly using the big core with peak frequency. This suggests that GreenWeb can adapt to events with different QoS targets.

For all events whose QoS type is “continuous,” we see a large difference in energy savings between the imperceptible and usable scenarios. This suggests that in general GreenWeb must spend a substantial amount of time on the big core in order to meet the imperceptible QoS target (16.6 ms), but it can use little core configurations most of the time to meet the usable QoS target (33.3 ms).

QoS Violation QoS violation is defined as the percentage by which a frame latency exceeds the QoS target. For example, a frame latency of 200 ms leads to an 100% QoS violation under a 100 ms QoS target. For events with a “continuous” QoS type, we report the geometric mean of all associated frames. It is worth noting that although *Perf* behaves the same under the two usage scenarios, its QoS violations are different because the QoS targets are different.

Figure 6.6b shows the QoS violation of GreenWeb on top of *Perf*. On average, GreenWeb introduces 1.3% and 1.2% more QoS violations than *Perf* under the imperceptible and usable usage scenario, respectively. In the imperceptible mode, three application events (MSN, LZMA-JS and BBC) whose QoS type is “single” have relatively high QoS violations. The high QoS violation is primarily introduced by GreenWeb's profiling runs to construct the predictive models (see Chapter **??**). For example, MSN's interaction requires peak performance to minimize QoS violations. GreenWeb takes two profiling runs, one of which is using the minimum frequency, to adapt to the peak performance. The minimal frequency run causes significant QoS violations. In contrast, events that have a "continuous" QoS type trigger a large amount of frames. Therefore, the profiling overhead is effectively amortized, and their QoS violations are negligible.

Some events that have a "continuous" QoS type have relatively high QoS violations under the usable mode. Outstanding examples are W3School and Cnet. Our analysis shows that most of the QoS violations come from frame complexity surges in a continuous frame sequence. `GreenWeb` aggressively scales down performance when the QoS target is low, and did not always react to the sudden frame complexity increase quickly. This suggests that the `GreenWeb` runtime could be better enhanced to capture the pattern of frame fluctuation in an event, potentially through offline profiling [133].

6.6.2 Annotation Effort

It is important to keep the annotation process lightweight because a `GreenWeb`-based system requires Web applications to be explicitly annotated. Manually annotating the QoS information of each event is time consuming for two reasons. First, real Web applications typically contain tens or even hundreds of events, as shown in Table 6.3. Second, one would have to understand the event callback semantics to determine the QoS type and QoS target of each event, which makes it difficult to annotate unfamiliar code, which in turn makes `GreenWeb` impractical to use in real Web development.

Through our experience with evaluating `GreenWeb` on real Web applications, we find that the best practice is to use a combination of AutoGreen and manual annotation. We use AutoGreen because it greatly improves productivity. Throughout all benchmarked applications, AutoGreen finishes annotations under 1 minute for an entire annotation.

The downside of AutoGreen is that it does not always annotate QoS targets correctly because it conservatively assumes short response latency for events with a "single" QoS type (see discussion in Chapter 6.4). Therefore, we manually correct the QoS target for events that should have a long response latency. Overall, we find that the percentage of events whose QoS type is "single" is below 20% for all applications. Most application events have a QoS type of "continuous", corroborating the prevalence of animation in today's Web applications. The "Annotation" column in Table 6.3 shows that in the end we annotate over 50% of all events in most applications. Unannotated events are not directly triggered by mobile user interactions and therefore are not the optimization target of `GreenWeb`, as discussed in Chapter 5.2.

Overall, it took us about 5 ~10 minutes to annotate each application with the combined manual and automatic approach. While we are not familiar with each application's codebase, the annotation is not a labor-intensive process. We expect the overhead to be even lower for experienced developers who are more familiar with their own applications.

6.7 Discussion

Manual Annotation vs. Runtime Mechanism As an alternative to receiving QoS annotations (e.g., using `GreenWeb` APIs) from developers, the Web runtime could try to detect QoS information at runtime without language hints. One implementation is to implement the exact logic of AutoGreen within the Web runtime. However, there are three major drawbacks of such a runtime-based approach.

First, implementing the QoS detection at runtime is not scalable. For example, whenever the Web standard introduces a new method of implementing animation (i.e., "continuous" QoS type) the browser runtime has to be extended to support it. In contrast, with developers directly specifying the QoS type the runtime can confidently optimize for the "continuous" QoS type without having to know how an animation is implemented. Second, a pure runtime strategy cannot precisely detect the QoS target information of an event for exactly the same reason that AutoGreen cannot precisely detect QoS target. Third, detecting QoS at runtime also introduces runtime performance and energy overhead that could potentially offset the energy saving benefits.

Effectiveness in a Multi-application Environment The ACMP-based `GreenWeb` runtime implementation presented here assumes that all CPU resources in a SoC are available to the foreground Web application during scheduling. We believe that this ACMP-based runtime design is also applicable when multiple mobile applications are concurrently consuming CPU resources. The reason is two-fold.

First, today's ACMP systems have ample CPU resources, e.g., four big and four small cores in the Exynos 5410 SoC with each core cluster having DVFS capability. If there is a background application occupying some CPU resources, the `GreenWeb` runtime system will still have a large trade-off space to schedule, although with fewer resources. Second, today's mobile SoCs are on the verge of supporting fine-grained power management techniques such as per-core DVFS using integrated voltage regulators (IVRs) [117]. Therefore, the scheduling space will become larger to further accommodate concurrent applications in the near future.

Defense Against Mis-annotation One potential vulnerability of exposing `GreenWeb` hints to developers is that developers might place hints that lead to inefficient system decisions. For example, a developer could set every event's QoS target to an extremely low value, which causes the Web runtime always to operate at the highest performance with maximal energy consumption. Such a mis-annotation could be introduced either inadvertently as a program energy bug or intentionally as an adversarial attack.

The notion of user-agent intervention (UAI) [94] in the Web community can be used to defend against such an issue. In short, UAI contends that a Web platform should correct application behaviors that are deemed harmful or undesired. Most of today's Web runtime systems have already implemented plenty of UAI policies such as blocking malicious third-party scripts or re-prioritizing resource loading order under latency/bandwidth constraints. Similarly, a Web runtime could adopt a `GreenWeb`-specific UAI policy. One candidate is to specify an energy budget of any Web application and ignores overly aggressive `GreenWeb` annotations once the energy budget is consumed. We leave it as future work to define, express, and implement such a UAI policy.

Composability of QoS Abstractions Although the QoS type and QoS target abstractions are sufficient for expressing predominant QoS specifications on *today*'s mobile devices, in the long term we will see new user interaction forms (e.g., using visible light [128]) and new ways that users assess QoS experience. Therefore, it is important to design "primitive" QoS abstractions, based on which complex, higher-level QoS abstractions can be easily composed.

The composability of QoS abstractions is critical because enumerating every single possible kind of QoS experience in a Web programming model is not scalable. Instead, the Web programming model should ideally provide a QoS primitive library that lets developers construct different QoS specifications in a completely customized way. To achieve this goal will likely involve extensively surveying future human-computer interaction forms and new QoS specifications. We leave it for the next phase of research.

6.8 Related Work

We first discuss `GreenWeb` in the context of prior work on language support for Web performance (Chapter 6.8.1). Although there exists little prior work on language support for Web energy-efficiency, language extensions for general energy optimizations do exist, which we compare and contrast

GreenWeb against in (Chapter 6.8.2). Finally, we discuss why the two QoS abstractions we propose are more comprehensive than prior work on mobile QoS characterizations (Chapter 6.8.3).

6.8.1 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 [93], 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 [15], 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.

6.8.2 Language Support for Energy Efficiency

Language support for energy efficiency has recently become a major research thrust. Most work targets sensor-based applications and approximate computing whereas GreenWeb, to the best of our knowledge, is the first to focus on Web applications. In addition, most previously proposed language systems require developers to annotate applications manually. We show that GreenWeb annotations can be automatically applied without programmer intervention. We now compare GreenWeb with prior language proposals in greater detail.

Eon [165] provides language constructs that let developers express alternative program control flow paths and associate energy states with control flows, based on which the runtime selects control flow paths that are suitable given the device energy level. Green [60] provides APIs that let developers specify multiple approximate versions of a function and QoS loss constraints, which guide the runtime to save energy without violating QoS. Both proposals rely on developers supplying alternative implementations, which is an optimization not immediately applicable to Web applications. In the future, however, it would be interesting to evaluate such an optimization strategy in the Web domain.

Energy Types [81] and EnerJ [157] take the language support for energy-efficiency a step further by designing general type systems. Both work ensures sound and safe energy optimizations by enforcing static type checking. Both type systems target imperative programming, and therefore are not immediately applicable to Web programming which is inherently declarative. In the future, however, it would be interesting to study how to decorate DOM elements with different type qualifiers to guide the energy optimizations.

LAB [116] 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.

6.8.3 Mobile QoS Characterization

The two QoS abstractions and the design of GreenWeb APIs is driven by understanding application QoS requirements from an application events perspective, which is not the focus in the majority of existing mobile workload suite and benchmark [5, 8, 9, 18, 24, 27, 34, 39, 113, 148, 155]. Other benchmarks consider only a particular form of QoS. For example, BBench [101] considers the webpage load time as the QoS constraint for Web browsing; the Web latency benchmark [40] considers the event latency of user actions to webpage elements; the GFXBench [19] and BaseMark [7] benchmarks consider FPS. However, our QoS abstractions lead to a general methodology to identify QoS requirements of a wide range of applications.

Chapter 7

Retrospective and Prospective Remarks

This chapter provides retrospective and prospective views of my dissertation work. The retrospective part (Chapter 7.1) distills three principles developed from my work on building a high-performance while energy-efficient mobile Web computing system. The prospective part (Chapter 7.2) suggests next steps for generalizing the principles and outlines lucrative research items.

7.1 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. Overall, it demonstrates three general principles:

- Empowering Web Developers Without Overburdening Them
  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 prin-

ciple in my work is that developers should be integrated into the energy optimization loop. `GreenWeb` (Chapter 6) demonstrates a first step by empowering developers to express user QoS expectations as program annotations. The QoS annotations guide the runtime to make a calculated trade-off between performance and energy consumption. Meanwhile, to incentivize a wide usage of `GreenWeb`-like annotations, it is critical to ease developers' manual annotation efforts. AutoGreen framework explores the feasibility of automatic annotation with promising results.

- Exposing Architecture Details Without Losing Generality Mobile system-on-chips (SoC) undergo rapid design iterations with each generation incorporating more complex cores and IP blocks. A Web runtime system must embrace, but does not overly couple with, the unprecedented hardware complexity. A guiding principle for system designers is that the runtime/architecture interface should be enhanced with new abstractions—abstractions that expose enough hardware details while not imposing too strict constraints on runtime implementation. `WebRT` (Chapter 5) is a concrete example of this principle where processor core type and core frequency in an ACMP architecture are exposed as two new abstractions to the Web runtime. The two new hardware abstractions enable a large performance-energy scheduling space while do not impose any constraints how the scheduling should be implemented.

- Balancing Programmability With Domain Specialization Ultimately mobile processor architecture designs will have to improve both the performance and energy-efficiency simultaneously, not just making trade-offs between the two design goals. Architectural specialization comes as the 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` (Chapter 4) demonstrates one approach of balancing programmability and specialization by starting from a (well-customized) general-purpose design and incrementally incorporating modest specializations for the most lucrative software targets.

7.2 Prospective

The future of mobile Web computing is undoubtedly exciting. It is exciting not only because of the intellectual challenges it poses, but also because of its applicability to our everyday life and its profound impact on our society. Guided by the principles described in the previous chapter, I outline a few open problems that are critical to the next era of mobile Web computing.

- Composability of QoS Abstractions As mobile application domains such as virtual reality and augment reality become more mainstream, new forms of user interaction (e.g., through nose [50], visible light [128]) will emerge and as such users will have new ways of assessing their QoS experience. Although the QoS type and QoS target abstractions proposed in this work are sufficient for expressing QoS specifications in *today*'s mobile applications, it is not clear how to express a new QoS abstraction in the context of *future* mobile applications.

To express semantics of new QoS specifications, the next step of Web language research should understand how to design "primitive" QoS abstractions, based on which complex, higher-level QoS abstractions can be easily composed. The composability of QoS abstractions is critical because enumerating every single possible kind of QoS experience in a Web programming model is not scalable. Instead, the Web programming model should ideally provide a QoS primitive library that lets developers construct different QoS specifications in a completely customized way.

To achieve this goal will likely involve an intimate collaboration between the system and human-computer interaction community. Breakthroughs in neuropsychology research that seeks to construct fundamental human perception models will also play an important role.

- Scalable Web Runtime If today's hardware systems that a mobile Web runtime has to manage, e.g., an ACMP architecture, are already complex, the complexity in tomorrow's mobile hardware will be unprecedented. ITRS projects that the number of specialized IP blocks will reach upwards of 100X more than today by 2022. What further adds to the hardware complexity is the fragmentation issue where, for instance, one IoT device will have non-standard, vendor-customized, and application-specific hardware components that are not found in any other devices.

Future mobile Web runtime should effectively manage the hardware complexity in order to fully take advantage of the hardware capability. The challenge of the runtime design is one of scalability. On one hand, a monolithic design that appeals to all existing IP blocks is unscalable because the performance and energy overheads accumulate as the number of IP blocks surges. On the other hand, a completely customized runtime tailored for the specifics of a particular device is unscalable either. This is because software vendors would have to distribute different runtimes based on different device capabilities, essentially transferring the burden of managing hardware complexity to managing software complexity.

An ideal mobile Web runtime should strike a balance between the two extremes, allowing one piece of software to be distributed across all devices while providing flexibilities to support hardware components unique to each device. One promising approach is to borrow the principles of the microkernel-based OS design: a minimal runtime kernel equipped with extensible modules, each dealing with one or one group of hardware IPs. The extensibility, i.e., the ability to (un)load a runtime module with isolated concerns, is what makes this runtime design scalable. The research challenge is to carefully select what tasks go into the runtime kernel and to define what the abstractions at the kernel-module interface should be.

- Programmable Accelerators for Mobile Machine Learning

  Machine learning techniques are making a foray into mobile/edge computing, and will be an indispensable component in future mobile Web applications with the help of extensive library support (e.g., ConvNetJS [51]). Most research in the architecture community focuses on accelerating a rather narrow set of machine learning techniques (e.g., convolutional neural networks) and application domains (e.g., image processing) [**?**]. There is wider space of algorithms (e.g., unsupervised learning, recurrent neural networks) and application domains (e.g., speech, language) for which computational characteristics are yet to be well-understood and hardware solutions are yet to be found.

There is a need to design architectures that offer better performance and energy-efficiency on a broad set of machine learning tasks. The design should be guided by the same principle of `WebCore`, i.e., to balance general-purpose programmability with specialization. One particularly promising approach is to start by designing accelerator building blocks, which are then composed together to form a high-level architecture. The building blocks should exploit the fundamental computational characteristics and communication patterns common to various machine learning techniques. The composition process should allow flexible reconfiguration of different building blocks without losing much efficiency.

The idea is closely related to prior explorations of configurable processors (e.g. CCA [79], CGRA [96], LSSD [143]). However, the unique characteristics of machine learning tasks, especially the extremely high memory pressure, will most certainly yield new design insights and trade-offs.

Bibliography

[1] 3G/4G Wireless Network Latency: How did Verizon, AT&T, Sprint and T-Mobile Compare in July 2014? http://goo.gl/JTFvbg.

[2] 9 Causes of Bad App Reviews. http://goo.gl/ejQ69m.

[3] Android CPUFreq Governors. https://goo.gl/K6Ce4V.

[4] Android WebView APIs. http://goo.gl/kRR49d.

[5] Antutu benchmark. http://goo.gl/hNA8oa.

[6] Are Smartphones Getting Larger Because They Have To? http://goo.gl/EvrtDF.

[7] Basemark x. https://www.basemark.com/.

[8] Browsermark benchmark. http://goo.gl/XJVZu7.

[9] Browsing Bench. http://goo.gl/lDJ45s.

[10] CSS animationend Event. https://goo.gl/VKUdsA.

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

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

[13] CSS transitionend Event. https://goo.gl/G57QT5.

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

[15] CSS Will Change Module Level 1. http://goo.gl/V6jx5e.

[16] CSS3 Media Queries. http://www.w3.org/TR/css3-mediaqueries/.

[17] Document Object Model (DOM). http://www.w3.org/DOM/.

[18] Geekbench 3.0 benchmark. http://goo.gl/TJzTCW.

[19] Gfxbench benchmark. https://gfxbench.com/.

[20] Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology. In *Samsung Whitepaper*.

[21] HTTrack. https://www.httrack.com/.

[22] iOS Developer Library: UIWebView. https://goo.gl/x1bezW.

[23] Jank Busting for Better Rendering Performance. http://goo.gl/vILunD.

[24] Kraken benchmark. http:goo.gl/HoUDrE.

[25] LTE Subscriptions to Surpass 1 Billion This Year. http://goo.gl/bO8Sfs.

[26] NVidia: Adaptive VSync Technology. http://goo.gl/IvHym4.

[27] Octane benchmark 2.0. http://goo.gl/d2BxmQ.

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

[29] Optimizing your pages for speculative parsing. https://goo.gl/KnsNtN.

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

[31] R software. http://www.r-project.org.

[32] Rd2: “the three second rule”. http://goo.gl/pynBl.

[33] Speed, Performance, and Human Perception. http://goo.gl/5PbOnR.

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

[35] Survey: Exploring the Reasons Users Complain about Apps. http://goo.gl/270TOD.

[36] The Evolution of the Web. http://www.evolutionoftheweb.com/.

[37] Timing Control for Script-based Animations. http://goo.gl/hQdm6D.

[38] V-sync. https://en.wikipedia.org/wiki/Screen_tearing#V-sync.

[39] Vellamo benchmark. https://goo.gl/g9VRus.

[40] Web latency benchmark. http:goo.gl/B4fZEh.

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

[42] The Evolution of HTML5. http://goo.gl/y6E2d7, 2012.

[43] Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. In *ARM Whitepaper*, 2013.

[44] big.LITTLE Technology: The Future of Mobile. In *ARM Whitepaper*, 2013.

[45] Nvidia Tegra 4 Family CPU Architecture: 4-PLUS-1 Quad core. In *Nvidia Whitepaper*, 2013.

[46] Power Efficiency in OS X. https://goo.gl/60XEji, 2013.

[47] Smartphone Screen Sizes Keep On Growing—But Not For Much Longer. http://goo.gl/lOhVay, 2013.

[48] Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. In *Nvidia Whitepaper*, 2013.

[49] GPU Accelerated Compositing in Chrome. http://goo.gl/sIuf7g, 2014.

[50] Apple Watch Users Discover Another Way to Go "Hands Free". http://goo.gl/RnyujX, 2015.

[51] ConvNetJS. https://github.com/karpathy/convnetjs, 2016.

[52] Your Favourite App isn't Native. http://goo.gl/B7VABO, 2016.

[53] 7-cpu. ARM Cortex-A15 Specification. http://goo.gl/CXYook, 2017.

[54] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proc. of ISCA*, 2000.

[55] Alexa. Alexa. http://www.alexa.com/, 2017.

[56] ARM. Exploring the Design of the Cortex-A15 Processor. http://goo.gl/Pc8hPe, 2012.

[57] ARM. ARM DS-5. http://ds.arm.com/ds-5/optimize/, 2015.

[58] O. Azizi, A. Mahesri, B.C. 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.

[59] Carmen Badea, Mohammad R. Haghighat, Alexandru Nicolau, and Alexander V. Veidenbaum. Towards Parallelizing the Layout Engine of Firefox. In *Proc. of USENIX HotPar*, 2010.

[60] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. of PLDI*, 2010.

[61] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proc. of CODES+ISSS*, 2002.

[62] Battery University. Battery Statistics. http://goo.gl/90mMeb, 2011.

[63] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steven Swanson, and Michael Bedford Taylor. SiChrome: Mobile Web Browsing in Hardware to Save Energy. *DaSi: First Dark Silicon Workshop*, 2012.

[64] Joshua Bixby. 2012 Predictions: The Average Web Page Will Hit 1 MB, Google and Siri Will Face Off, and Chrome, Windows 7, and RUM will rise. http://goo.gl/WmcTsx, 2011.

[65] Joshua Bixby. The relationship between faster mobile sites and business kpis: Case studies from the mobile frontier. http://goo.gl/shnlDF, 2011.

[66] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *Proc. of HPCA*, 2013.

[67] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. The Model is Not Enough: Understanding Energy Consumption in Mobile Devices. In *Poster of HotChip*, 2012.

[68] Edward A Burton, Gerhard Schrom, Fabrice Paillet, James Douglas, William J Lambert, Krishnaja Radhakrishnan, and Michael J Hill. Fivr: Fully integrated voltage regulators on 4th generation intel core socs. In *Proc. of APEC*, 2014.

[69] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proc. of NSDI*, 2015.

[70] Michael Butler, Tse-Yu Yeh, Yalt Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single Instruction Stream Parallelism Is Greater than Two. In *Proc. of ISCA*, 1991.

[71] Ting Cao, Tiejin Gao, Stephen M. Blackburn, and Kathryn S. McKinley. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *Proc. of ISCA*, 2012.

[72] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer: An Information Workspace. In *Proc. of CHI*, 1991.

[73] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. of USENIX ATC*, 2010.

[74] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. of USENIX ATC*, 2010.

[75] Calin Cascaval, Seth Fowler, Pablo Montesinos Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. Zoomm: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of PPoPP*, 2013.

[76] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. EFetch: Optimizing Instruction Fetch for Event-driven Web Applications. In *Proc. of PACT*, 2014.

[77] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. Accelerating Asynchronous Programs through Event Sneak Peek. In *Proc. of ISCA*, 2015.

[78] Xiang Chen, Yiran Chen, Zhan Ma, and Felix CA Fernandes. How Is Energy Consumed In Smartphone Display Applications? In *Proc. of HotMobile*, 2013.

[79] Nathan T. Clark, Hongtao Zhong, and Scott A. Mahlke. Automated Custom Instruction Generation for Domain-Specific Processor Acceleration. In *IEEE Transactions on Computers*, 2005.

[80] Mark Claypool, Kajal Claypool, and Feissal Damaa. The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. In *Proc. of Multimedia Computing and Networking*, 2006.

[81] Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu. Energy types. In *Proc. of OOPSLA*, 2012.

[82] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling Heterogeneous Multi-cores through Performance Impact Estimation (PIE). In *Proc. of ISCA*, 2012.

[83] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le. RAPL: Memory Power Estimation and Capping. In *Proc. of ISLPED*, 2010.

[84] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion–Implanted MOSFET's With Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, 1974.

[85] Mian Dong and Lin Zhong. Chameleon: a color-adaptive web browser for mobile oled displays. In *Proc. of MobiSys*, 2012.

[86] G. Dunteman. *Principal Component Analysis.* Sage Publications, 1989.

[87] Kit Eaton. How 1s Could Cost Amazon $1.6 Billion in Sales. http://goo.gl/qG0M2Q, 2013.

[88] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proc. of HPCA*, 2009.

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

[90] Hadi Esmaeilzadeh, Emily Blem, Rene St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.

[91] Dave Evans. The Internet of Things: How the Next Evolution of the Internet is Changing Everything. In *Cisco Whitepaper*, 2011.

[92] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. Buffer-Integrated-Cache: A Cost-effective SRAM Architecture for Handheld and Embedded Platforms. In *Proc. of DAC*, 2011.

[93] Darin Fisher and Gagan Saksena. Link prefetching in mozilla: A server-driven approach. In *Web content caching and distribution*, pages 283–291. Springer, 2004.

[94] Dimitri Glazkov. User Agent Intervention. http://bit.ly/user-agent-intervention.

[95] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. In *IEEE Journal of Solid-State Circuits*, 1996.

[96] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. DySER: Unifying functionality and parallelism specialization for energy efficient computing. In *IEEE MICRO*, 2012.

[97] Ilya Grigorik. *High Performance Browser Networking.* O'Reilly, 2013.

[98] D. Grunwald, P. Levis, K. Farkas, C.B. Morrey III, and M. Neufeld. Policies for Dynamic Clock Scheduling. In *Proc. of OSDI*, 2000.

[99] Dirk Grunwald, Charles B. Morrey, III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *Proc. of OSDI*, 2000.

[100] Qi Guo, Tianshi Chen, Yunji Chen, Zhihua Zhou, Weiwu Hu, and Zhiwei Xu. Effective and Efficient Microprocessor Design Space Exploration Using Unlabeled Design Configurations. In *Proc. of IJCAI*, 2011.

[101] 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.

[102] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proc. of ISCA*, 2000.

[103] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Proc. of ISPASS*, 2015.

[104] 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. In *Proc. of HPCA*, 2016.

[105] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proc. of ISCA*, 2010.

[106] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proc. of ICSE*, 2013.

[107] Hardkernel. ODROID-XU+E Development Board. http://goo.gl/Ige0Jp, 2015.

[108] Frank E Harrell. *Regression Modeling Strategies*. Springer, 2001.

[109] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2009.

[110] Urs Hoelzle. The Google Gospel of Speed. https://goo.gl/fTd0f0, 2012.

[111] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. of MobiSys*, 2012.

[112] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proc. of MobiSys*, 2012.

[113] Yongbing Huang, Zhongbin Zha, Mingyu Chen, and Lixin Zhang. Moby: A Mobile Benchmark Suite for Architectural Simulators. 2014.

[114] Jos A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck Identification and Scheduling in Multithreaded Applications. In *Proc. of ASPLOS*, 2012.

[115] Jos A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. In *Proc. of ISCA*, 2013.

[116] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proc. of OOPSLA*, 2013.

[117] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proc. of HPCA*, 2008.

[118] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: an Energy Efficient Memory Structure. In *Proc. of MICRO*, 1997.

[119] Brian Klug and Anand Lal Shimpi. Krait cache and memory hierarchy. http://goo.gl/ZuO7X2, 2011.

[120] Theo Kluter, Philip Brisk, Edoardo Charbon, and Paolo Ienne. Way Stealing: A Unified Data Cache and Architecturally Visible Storage for Instruction Set Extensions. In *IEEE Transactions on VLSI*, 2013.

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

[122] Kssmetrics. Speed is a killer. http://goo.gl/4PfsJL, 2011.

[123] AV Kumar. *Mobile Computing Techniques in Emerging Markets: Systems, Applications and Services: Systems, Applications and Services.* IGI Global, 2012.

[124] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of MICRO*, 2003.

[125] Benjamin C. Lee and David M. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proc. of ASPLOS*, 2006.

[126] Paul Lewis. Rendering performance. https://goo.gl/Ff5HrD, 2014.

[127] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of MICRO*, 2009.

[128] Tianxing Li, Chuankai An, Zhao Tian, Andrew T. Campbell, and Xia Zhou. Human sensing using visible light communication. In *Proc. of MobiCom*, 2015.

[129] Yingmin Li, Mark Hempstead, Patrick Mauro, David Brooks, Zhigang Hu, and Kevin Skadron. Power and Thermal Effects of SRAM vs. Latch Mux Design Styles and Clocking Gating Choices. In *Proc. of ISLPED*, 2005.

[130] Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the Efficiency of Text Processing on

Commodity Processors. In *Proc. of HPCA*, 2012.

[131] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. SODA: A Low-power Architecture For Software Radio. In *Proc. of ISCA*, 2006.

[132] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proc. of MSR*, 2014.

[133] Daniel Lo, Taejoon Song, and G. Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *Proc. of MICRO*, 2015.

[134] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity into a Core. In *MICRO*, 2012.

[135] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proc. of ASPLOS*, 2012.

[136] Haohui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Montesinos Pablo. A Case for Parallelizing Web Pages. In *Proc. of USENIX HotPar*, 2012.

[137] Leo A. Meyerovich and Rastislav. Bodik. Fast and Parallel Webpage Layout. In *Proc. of WWW*, 2010.

[138] Leo A. Meyerovich and Rastislav Bodik. FTL: Synthesizing a Parallel Layout Engine. In *Proc. of ECOOP*, 2012.

[139] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Proc. of MICRO*, 2012.

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

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

[142] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

[143] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Pushing the Limits of Accelerator Efficiency while Retaining Programmability. In *Proc. of HPCA*, 2016.

[144] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proc. of Sensys*, 2013.

[145] OpenHub. Chromium Project Summary: Languages. https://goo.gl/XQb3EO, 2017.

[146] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Trace Based Phase Prediction For Tightly-Coupled Heterogeneous Cores. In *MICRO*, 2013.

[147] Venkatesh Pallipadi and Alexey Starikovskiy. The Ondemand Governor: Past, Present, and Future. In *Linux Symposium*, 2006.

[148] Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu. Performance, Energy Characterizations and Architectural Implications of an Emerging Mobile Platform Benchmark Suite-MobileBench. In *Proc. of IISWC*, 2013.

[149] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proc. of DAC*, 2011.

[150] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys*, 2012.

[151] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proc. of OOPSLA*, 2014.

[152] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution Engine: Bal-

ancing Efficiency & Flexibility in Specialized Computing. In *Proc. of ISCA*, 2013.

[153] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely., and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proc. of ISCA*, 2007.

[154] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. In *ISCA*, 2009.

[155] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated Construction of JavaScript Benchmarks. In *Proc. of OOPSLA*, 2011.

[156] Kunihiko Sakamoto. Time-to-first-x-paint metrics: Status and refinement plans. https://goo.gl/xyab3A, 2015.

[157] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proc. of PLDI*, 2011.

[158] Fred Schlachter. No moore's law for batteries. In *Proc. of National Academy of Science of the United States of America*, 2013.

[159] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. Livelab: Measuring wireless networks and smartphone users in the field. In *SIGMETRICS Performance Evaluation Review*, 2011.

[160] Ben Shneiderman. *Designing the User Interface.* Addison-Wesley, 1992.

[161] Open Signal. Android Fragmentation Visualized. http://goo.gl/ODlx4z, 2014.

[162] Shikhir Singh. HTML5 On The Rise: No Longer Ahead Of Its Time. http://goo.gl/yuEVCy, 2015.

[163] Mac Slocum. You can't get away with a bad mobile experience anymore. http://goo.gl/T3812z, 2011.

[164] Michael D. Smith. Overcoming the Challenges to Feedback-Directed Optimization (Keynote Talk). In *Proc. of DYNAMO*, 2000.

[165] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proc. of SenSys*, 2007.

[166] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proc. of ASPLOS*, 2009.

[167] M Aater Suleman, Yale N Patt, Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, and Doug Carmean. Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency. Technical Report TR-HPS-2007-001, The University of Texas as Austin, 2007.

[168] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proc. of WWW*, 2012.

[169] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Number HPL-2008-20, 2008.

[170] Po-Hsien Tseng, Pi-Cheng Hsiu, Chin-Chiang Pan, and Tei-Wei Kuo. User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices. In *DAC*, 2014.

[171] W3C. CSS Cascading Order. https://goo.gl/PkKg92, 2014.

[172] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are Web Browsers Slow on Smartphones? In *Proc. of HotMobile*, 2011.

[173] WebKit. Webkit. http://www.webkit.org, 2015.

[174] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proc. of OSDI*, 1994.

[175] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proc. of OSDI*, 1994.

[176] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. In *Proc. of ISCA*, 2009.

[177] Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, Dan Connors, David Brooks, Margaret Martonosi, and Douglas W. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proc. of MICRO*, 2005.

[178] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proc. of PLDI*, 2003.

[179] Ye Xu, Mu Lin, Hong Lu, Giuseppe Cardone, Nicholas D. Lane, Zhenyu Chen, Andrew T. Campbell, and Tanzeem Choudhury. Preference, Context and Communities: A Multi-faceted Approach to Predicting Smartphone App Usage Patterns. In *Proc. of ISWC*, 2013.

[180] Allan Yogasingam. Teardown: Samsung galaxy s4. http://goo.gl/BQl4Dg, 2013.

[181] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. Smart Caching for Web Browsers. In *Proc. of WWW*, 2010.

[182] Zhijia Zhao, Mingzhou Zhou, and Xipeng Shen. SatScore: Uncovering and Avoiding a Principled Pitfall in Responsiveness Measurements of App Launches. In *Proc. of UbiComp*, 2014.

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

[184] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. The Role of the CPU in Energy-Efficient Mobile Web Browsing. In *Micro, IEEE*, 2015.

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

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

[187] Yuhao Zhu and Vijay Janapa Reddi. GreenWeb: Language Extensions for QoS-aware Energy-Efficient Mobile Web Computing. In *Proc. of PLDI*, 2016.

[188] Yuhao Zhu and Vijay Janapa Reddi. Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing. In *In Submission*, 2016.

[189] Yuhao Zhu, Aditya Srikanth, Jingwen Leng, and Vijay Janapa Reddi. Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing. In *Computer Architecture Letters*, 2014.

Vita

Yuhao Zhu was born in Wuxi, Jiangsu, P.R. China, in 1988. He received the Bachelor of Science degree in Computer Science and Engineering from Beihang University (previously known as Beijing University of Aeronautics and Astronautics), Beijing, China in 2010. He worked as an undergraduate researcher in the last year of the college at Tsinghua University, working with Yangdong Deng on GPGPU, parallel EDA algorithms, and IP router microarchitectures. He is now a graduate researcher and Ph.D. candidate at the Electrical and Computer Engineering Department in The University of Texas at Austin, Austin, TX, working with Vijay Janapa Reddi. He is on track to graduate in May 2017. Since Aug 2016, he has also been a Research Fellow visiting the School of Applied Engineering and Science in Harvard University, Cambridge, MA. His Ph.D. research has been supported by the Google Ph.D. fellowship and the Microelectronics and Computer Development Fellowship at UT Austin. He was a research intern/Co-op engineer at STMicroelectronics, AMD Research, and Google Inc.

Permanent address: 1630 W 6th St.
Austin, Texas 78703

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.