High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems

Yuhao Zhu and Vijay Janapa Reddi

Department of Electrical and Computer Engineering
The University of Texas at Austin
yuhao.zhu@austin.utexas.edu, vj@ece.utexas.edu

Abstract

Internet web browsing has reached a critical tipping point. Increasingly, users rely more on mobile web browsers to access the Internet than desktop browsers. Meanwhile, webpages over the past decade have grown in complexity by more than tenfold. The fast penetration of mobile browsing and ever-richer webpages implies a growing need for high-performance mobile devices in the future to ensure continued end-user browsing experience. Failing to deliver webpages meeting hard cut-off constraints could directly translate to webpage abandonment or, for e-commerce websites, great revenue loss. However, mobile devices' limited battery capacity limits the degree of performance that mobile web browsing can achieve. In this paper, we demonstrate the benefits of heterogeneous systems with big/little cores each with different frequencies to achieve the ideal trade-off between high performance and energy efficiency. Through detailed characterizations of different webpage primitives based on the hottest 5,000 webpages, we build statistical inference models that estimate webpage load time and energy consumption. We show that leveraging such predictive models lets us identify and schedule webpages using the ideal core and frequency configuration that minimizes energy consumption while still meeting stringent cut-off constraints. Real hardware and software evaluations show that our scheduling scheme achieves 83.0% energy savings, while only violating the cut-off latency for 4.1% more webpages as compared with a performance-oriented hardware strategy. Against a more intelligent, OS-driven, dynamic voltage and frequency scaling scheme, it achieves 8.6% energy savings and 4.0% performance improvement simultaneously.

1. Introduction

Mobile web browsing is shifting the balance of Internet traffic. Recent statistics indicate that users now spend 50.2% of their time using the web browser [1], causing mobile Internet traffic to surpass the amount of desktop Internet traffic in major regions of the world [2]. The proliferation of mobile devices and social networks is fueling this trend. Meanwhile, web monetization (i.e., the average revenue per user) is still 5X lower on mobile than desktop [1]. Media and advertisement providers are harnessing this opportunity, and thus causing an even-faster penetration of mobile web browsing.

Accompanying the critical mass of the mobile market is the trend that webpages are becoming significantly more complex and computationally intensive over the past decade. For example, Fig. 1 shows the network transmission time and content processing time for `www.cnn.com` over the past 11 years. We randomly pick one image from each year archived by Internet Archive [3], and measure on a dual-core ARM Cortex-A9 mobile processor connected to a 100Mb/s Ethernet. The trend-line in the plot indicates a tenfold relative increase in webpage computational intensity from the perspective of compute versus network.

As the computational intensity of webpages increases, there is growing concern over webpage loading time. A recent investigation concluded that users tend to abandon webpages that do not load within a certain cut-off latency [4]. Related statistics on e-commerce websites reveal that a 1-second delay in webpage load could result in a $2.5 million loss in sales per year [5]. The hard-constraint of cut-off latency implies a need for high-performance mobile devices. However, energy constraints limit us from achieving desktop-level compute capability on mobile devices. Therefore, the challenge is to ensure high performance while minimizing the energy consumption. Owing to a large scheduling space, heterogeneous systems with big/little cores each with DVFS capabilities enable a flexible trade-off between performance and energy.

In this paper, we demonstrate the benefits of big/little heterogeneous systems in achieving an ideal performance and energy trade-off via scheduling. We harness the key insight that different webpages contain strong variances in load time and energy consumption. Leveraging this fact, we propose webpage-aware scheduling, a mechanism that explores the webpage variance and intelligently schedules webpages using different core and frequency configurations for minimizing the energy while still meeting the cut-off constraint.

Specifically, Fig. 2 shows the load time and energy consumption of six hot webpages loaded on the Cortex-A9 at 1.2 GHz. We observe strong variances across the webpages. For example, `www.cnn.com` takes approximately 6 seconds to load while consuming 8.9 Joules of energy. In contrast, `www.craigslist.org` takes less than 1 second to load while consuming only 1.6 Joules of energy. Detailed characterizations reveal that such variance is the result of inherent variances of webpage primitives (e.g., HTML, CSS). Regression modeling techniques capture the inherent webpage variance, and let us estimate the webpage load time and energy consumption under different core and frequency combinations.

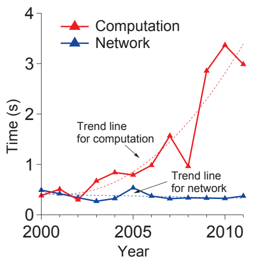


Fig. 1: Increasing computational intensity of webpages.

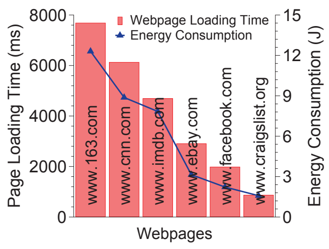


Fig. 2: Load time and energy consumption of webpages.

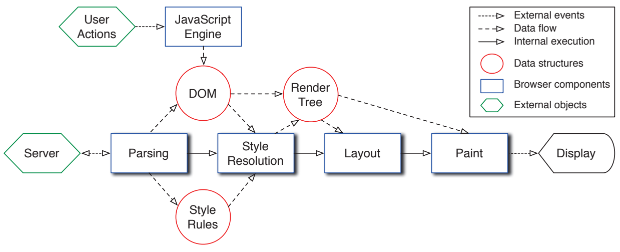


Fig. 3: Overview of the web browser architecture.

On the basis of the estimations, we predict the ideal execution core and frequency for webpages and schedule them accordingly to satisfy the cut-off constraint with the least energy. In summary, we make the following contributions in this paper:

1. We explore the energy-delay trade-off of big/little systems for mobile web browsing and demonstrate the potential benefits (Sec. 4).
2. We identify the root cause of webpage variance by characterizing and quantifying different aspects of webpage primitives (Sec. 5).
3. We show that webpage load time and energy consumption can be predicted via regression models (Sec. 6).
4. We demonstrate how to leverage a big/little system effectively via the proposed webpage-aware scheduling (Sec. 7).

Measured hardware results, of a synthesized big/little system based on the ARM Cortex-A9 and A8 processors, demonstrate that as compared with a performance-oriented hardware strategy, webpage-aware scheduling achieves 83.0% energy savings, while only violating the cut-off latency for 4.1% more webpages. Against a more intelligent OS-driven DVFS strategy, webpage-aware scheduling achieves 8.6% energy reduction while shortening load time by 4.0% on average.

2. Mobile Web Browsing

We explain what is important for end users during the mobile web browsing experience (Sec. 2.1). Following that we provide a brief overview of how a web browser works, and we identify potential sources of computational intensity (Sec. 2.2).

2.1. The Mobile Web Browsing Experience

A neurological study conducted in 2010 by the CA Technologies concluded that poor web browsing experience can lead to “web stress” that causes users to use a rival website or abandon the transaction altogether [4]. Google engineers measured this effect, stating that “a 400 ms delay leads to a 0.44% drop in search volume” [6]. With over 2.27 billion Internet users in the world [7], this seemingly negligible drop can translate to approximately 9.98 million unsatisfied users.

The most important criteria for mobile browsing experience is the webpage load time. It is the primary cause of webpage abandonment [5]. Statistics indicate that 47% of mobile web consumers expect a webpage to load in 2 seconds or less, and beyond that cut-off latency the rate drops by 6.7% for each additional second. A 4 second webpage load time can translate to a 25% increase in webpage abandonment [5].

Furthermore, in a world where mobile web-based e-commerce is rapidly emerging as the new means of conducting online sales, missing the cut-off load time can have significant financial ramifications to institutions [1]. A recent study concluded that an e-commerce website generating a revenue of $100,000 per day stands to lose up to $2.5 million in sales each year for every second of delay in webpage load [5].

2.2. The Web Browser Architecture

At the center of the web browser architecture is the rendering engine, which processes the webpage documents returned from the server and displays them on the screen. After the webpage is loaded, users can dynamically interact with the webpage through JavaScript. In this paper, we focus only on the rendering engine because it is the first determinant part of the mobile web browsing experience. Dynamic webpage interaction is beyond the scope of this work, and we refer readers to Sec. 8 for a discussion about JavaScript. Using Fig. 3, we present an overview of the web browser architecture.

The rendering engine starts with *parsing* HTML and CSS documents. HTML contains the text and overall skeleton of a webpage through a combination of tags, each associated with zero or more attributes. Along with parsing the HTML file, the rendering engine dynamically constructs the *DOM* (Document Object Model) *tree* data structure, in which each node corresponds to a HTML tag, an attribute, or a section of text. The DOM tree can be viewed as an abstract and concise representation of the webpage structure.

Complementary to HTML, CSS determines the webpage’s visual style information through a set of *style rules*, each with a selector and a set of properties. Each rule is intended to select one or a group of HTML tags to apply the properties. During parsing, the rendering engine extracts CSS rules and constructs the corresponding data structure.

Given the DOM tree and CSS rules, the *style resolution* module determines the webpage’s style information (e.g. color, font) by constructing the *render tree*. The render tree is the visual representation of a webpage, with each node corresponding to one visual element in the webpage. Thereafter, the *layout* module operates on the render tree to calculate the exact coordinates on the screen for each visual element. The *painting* module then walks the render tree and calls graphic libraries to perform the actual display of the webpage.

Despite the optimizations that web browsers perform to improve style resolution, layout, and painting, they are still the most time-consuming components [40]. The stages' computations inherently involve intensive tree operations that are difficult to parallelize, underutilize the hardware resources, and dissipate energy inefficiently [42, 45].

3. Experimental Setup

Platform(s) We use the Cortex-A9 mobile processor as the big core. All big-core-related experiments are conducted on the Pandaboard ES (rev. B1) running Ubuntu 12.04 with Linux kernel version 3.2.14. It comes with a market-quality OMAP 4460 system-on-chip (SoC) equipped with a dual-core A9 processor manufactured on the 45 nm process node. The A9 is a complex out-of-order four-wide superscalar core with eight pipeline stages. It has 32 KB L1 I/D caches and a 512 KB L2 cache. It can operate at 300 MHz, 700 MHz, 920 MHz or 1.2 GHz, with a measured core voltage of 0.83 V, 1.01 V, 1.11 V or 1.27 V, respectively.

We choose a low(er)-power Cortex-A8 as the little core. Little-core experiments are performed on the BeagleBoard-xM development board (rev. C1) equipped with the DM 3730 SoC (OMAP3 series) that encapsulates an A8 core. The A8 processor is also manufactured on the 45 nm process node. It is an in-order, dual-issue processor with 13 pipeline stages and 32 KB I/D caches and a 256 KB L2 cache. It can operate at 300 MHz, 600 MHz, or 800 MHz, with a measured core voltage of 0.94 V, 1.10 V, or 1.26 V, respectively.

Webpages We consider the top 5,000 webpages in the Internet ranked by `www.alexa.com`. These webpages cover a wide distribution of website categories from business to news, shopping, search engines, and so forth. Because a large portion of these websites do not have a mobile version, for fair comparison, we use the desktop version for all the webpages.

Load time Measurement We instrument Mozilla Firefox 12.0's rendering engine for measurements, eliminating the browser's bootstrap and shut-down effects. Since we study individual webpages, we do not consider the browser's multitab feature. To isolate and study each webpage's behavior closely, we disable the browser cache to avoid pollution of partially cached webpage resources. Our measurement resolution is in the order of microseconds.

In addition, motivated by Fig. 2, we focus only on computation and isolate network and disk overhead by downloading the webpages and mapping them into RAMFS to keep the entire working set in memory. In fact, previous work showed that the cache can largely capture the working set, such that the delay of network and I/O is greatly cushioned [32]. We verify that but do not include the data due to space limitations.

Energy Measurement We built a power-sensing circuitry (Fig. 4) using a sense resistor located before the off-chip voltage regulator module (VRM) [8] to measure the A9's energy consumption. Using National Instruments' DAQ unit X-series 6366, we gather power measurements by simultaneously sens-

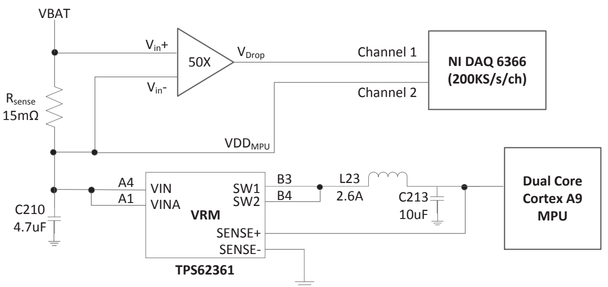


Fig. 4: Power measurement circuitry.

ing the voltage drop across a current shunt resistor of 15 $m\Omega$, as well as the V_{DD} to the processor, both at a rate of 200,000 samples per second. We must keep an extremely small voltage drop across the shunt resistor to preserve the original processor current draw characteristics, so we use a Texas Instruments INA199A1-A3EVM module with an INA199A1 current shunt amplifier to achieve a gain of 50X. The resolution of our measurement is 78 µV. We use a similar setup for measuring the power of A8 in the DM 3730 SoC.

The run-to-run variance in the experimental measurement setup is negligible. We verify the setup's reproducibility by measuring the load time and energy variance across 10 runs for the benchmarked webpages. Variance is within 3%.

4. Motivation: Energy-Delay Trade-off

We aspire to answer a fundamental question: does webpage loading benefit from big/little heterogeneous systems? For example, can the processor lower the frequency for a simple webpage to consume less energy but still respect the cut-off constraint? Can a webpage originally scheduled on the (energy-consuming) big core be migrated to the (energy-saving) little core without violating the cut-off constraint?

On the basis of the detailed measurements and analysis on the 5,000 webpages, we find that different webpages require different core and frequency configurations to meet the latency cut-off constraint while minimizing the energy. This suggests that a heterogeneous system with both a big core and a little core, each capable of performing DVFS, is strongly beneficial.

To demonstrate the benefits of such heterogeneous systems, we measure the webpage load time and energy consumption of all 5,000 hot 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, and we subsequently expand our analysis to include the comprehensive set of all webpages.

Representative analysis Fig. 5 shows the energy versus delay plots for the four representative webpages. Assuming 3 seconds as the cut-off latency for webpage load [9], 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

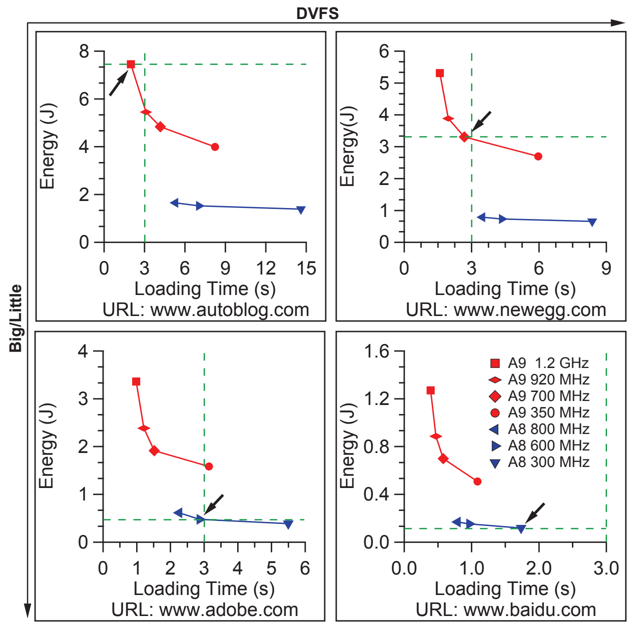


Fig. 5: Webpages have different ideal execution configurations to meet the cut-off latency while consuming the least energy.

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.

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.

Comprehensive analysis We extend our analysis to the full set of 5,000 webpages. Fig. 6 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 Fig. 6 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 [9], 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 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.

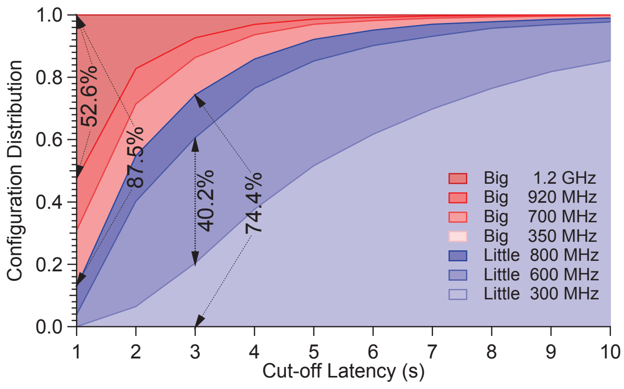


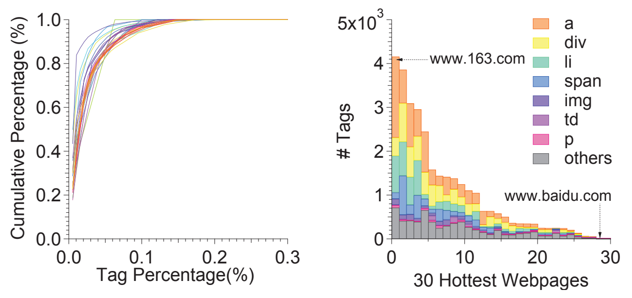
Fig. 6: The distribution of ideal core and frequency configurations under different cut-off latencies.

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 Fig. 6 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. Webpage Characterization

The key to effectively harnessing a big/little system's benefits is to understand the root cause of load time and energy consumption variance across different webpages. In this section, we demonstrate that the root cause of webpage variance arises from the inherent "webpage variance" in the structural (HTML) and style (CSS) information.



(a) HTML tag cumulative distribution. (b) HTML tag mix.

Fig. 7: HTML tag analysis.

We mine 5,000 hot webpages and apply a combination of static profiling of real webpages and runtime measurement on microbenchmarking webpages. Static profiling shows that different webpages have vastly different distributions of webpage primitives. Runtime microbenchmarking reveals that different webpage primitives have significantly different execution overhead and energy consumption. Due to the space constraints, we only show data for the 30 hottest webpages; still, webpage variance is still strongly observable.

5.1. HyperText Markup Language (HTML)

The HTML document consists of tags and attributes, and it is represented as a DOM-tree data structure that plays an important role in webpage rendering. We characterize the tag, attribute, and DOM tree separately.

Tags In a webpage, HTML tags describe a webpage's fundamental functionality. We study both the static and runtime characteristics of HTML tags. Under static analysis, we perform a cumulative distribution profiling of the HTML tag usage and show the results in Fig. 7a. The y-axis represents the cumulative percentage of tags that appear in a webpage. The x-axis begins with the hottest tag and descends toward the least-hot tag across all webpages. Only 10% of all HTML tags ($\sim$14) make up nearly 90% of the entire tag usage, indicating the existence of strong "tag" locality across webpages. We infer that web developers are used to utilize a few common HTML tags.

To better understand how tags vary across different webpages, we perform tag-mix profiling, similar to conventional instruction-mix profiling. In Fig. 7b webpages are sorted by the total number of tags from left to right. We make three important observations. First, a few tags are hot across all webpages. We group the rest together into the "others" category in the figure. Second, the number of tags varies significantly across webpages. For example, `www.baidu.com`, which is the 5th hottest website, has only 126 tags, whereas `www.163.com`, which ranks 28th, has 4,135 tags. Finally, the hot tags are not always evenly hot in all webpages; for instance, `www.baidu.com` does not have the <td> tag, which is present in nearly all other webpages.

We conduct load time and energy microbenchmarking of HTML tags to quantify the runtime behavior of different

Category	Microbenchmark	Time (ms)		Energy (mJ)	
		Abs.	Norm.	Abs.	Norm.
Tag	h3	8.85	1.0	26.31	1.0
	li	14.25	1.6	44.97	1.7
	option	21.48	2.4	67.12	2.6
	table	20.96	2.4	69.82	2.7
	input	44.92	5.0	166.28	6.3
	img	177.03	20.0	611.53	23.0
Attribute	cellspacing	17.30	1.0	63.45	1.0
	border	23.46	1.4	81.87	1.3
	bgcolor	28.05	1.6	97.86	1.5
	background	337.18	19.0	1145.75	18.0
DOM tree node count	1k	325.65	1.0	1103.98	1.0
	2k	560.73	1.7	1897.46	1.7
	4k	1017.19	3.1	3451.76	3.1
	8k	2041.87	6.3	6903.21	6.3

Table 1: HTML Microbenchmarking Results

HTML tags. We choose a subset of hot tags that can be visually seen in webpages. Each microbenchmark is a simple webpage repeating one particular HTML tag 100 times, except the <img> microbenchmark which only loads once a Portable Network Graphics (PNG) image of 185 KB (average image size per webpage as reported in [10]). By observing differences in the load time and energy consumption between microbenchmark webpages and a blank baseline webpage, we compute the execution time and energy consumption associated with the particular HTML tag. We use the same microbenchmarking methodology in the following characterization sections, unless stated as otherwise.

From the microbenchmarking results shown in the "tag" category in Tbl. 1, we observe different execution overhead and energy consumption from the different tags. For example, the <input> tag used for inputting characters takes 5X execution time compared to a simple <h3> header tag. The <input> tag also consumes 6.3X more energy than the <h3> tag.

Attributes Auxiliary to HTML tags are attributes that delimit the functionality of tags and specify extra information for them. We perform similar static attribute profiling, observing that the number of attributes varies significantly across webpages. Of all the attributes, layout-related attributes are the most important. They can lead to partial relayout of the webpage, introducing extra computations. Fig. 8 shows the distribution of the attributes that affect webpage layout. We observe a wide distribution across webpages. The last 10 websites use fewer than 100 attributes to style the webpage. Consequently, they are less intrusive to webpage rendering.

To quantify the intrusiveness of different layout-related attributes to webpage rendering, we conducted microbenchmarking. The "attribute" category in Tbl. 1 shows the representative subset of the results. Except the background microbenchmark, we use a webpage containing 5,000 1 $\times$ 1 HTML tables without any attributes as the baseline. Each microbenchmark applies one particular layout-related attribute to the table, and repeats it 1,000 times. The background microbenchmark loads

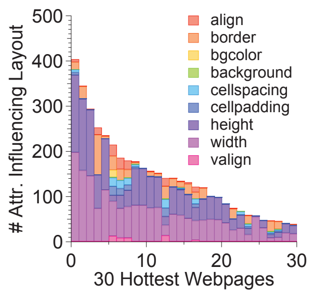


Fig. 8: Distribution of attributes that influence webpage layout.

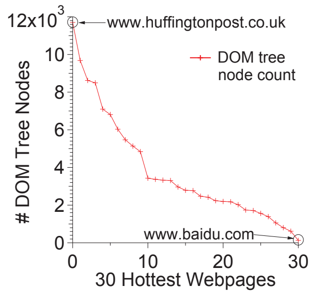


Fig. 9: DOM tree node distribution.

a 185 KB PNG image as the webpage background and uses a blank webpage as the baseline.

Similar to HTML tags, attributes also exhibit different execution overhead and energy consumption. For example, processing the background attribute is 19X slower and 18X more energy-consuming than processing 1,000 cellspacing attributes.

DOM Tree The DOM tree is semantically equivalent to an HTML document. As described in Sec. 2.2, the rendering engine operates on the DOM tree intensively for style resolution, layout, etc. Therefore, the DOM tree's size is a strong heuristic indicative of a webpage's processing complexity.

Webpages have vastly different DOM-tree sizes in terms of the number of DOM-tree nodes as shown in Fig. 9. We highlight the difference in webpages' DOM tree sizes by picking two extreme cases (`www.huffingtonpost.co.uk` and `www.baidu.com`). As the figure shows, the difference in DOM tree size between the two webpages is over 1,200X.

We perform microbenchmarking to quantify the impact of DOM-tree size on load time and energy consumption. We vary the number of DOM-tree nodes in each microbenchmarking experiment, and we compare webpage load time and energy consumption against a blank baseline webpage. The "DOM-tree node count" category in Tbl. 1 shows the results. Microbenchmarking shows that the execution time and energy consumption increase with the number of DOM-tree nodes. We observe a nearly linear relationship between DOM-tree size and its rendering time and energy consumption.

We also study DOM-tree depth and the average fanout for each node in the DOM tree. However, our analysis indicates that those metrics have less significant impact on load time and energy consumption than DOM-tree size.

5.2. Cascading Style Sheet (CSS)

CSS is at the center of the style resolution and layout modules. Most modern webpages heavily use CSS to specify rules regarding how and where HTML tags should be presented in a webpage. CSS rules use selectors to select HTML tags, and can apply to them different properties such as color, formatting, floating preference, etc. We therefore characterize CSS for both selectors and properties.

Selector The complexity associated with selector processing comes primarily from two aspects: the total number of selectors and the selection pattern of selectors. The number

Category	Microbenchmark	Time (ms)		Energy (mJ)	
		Abs.	Norm.	Abs.	Norm.
Selector count	1k	9.09	1.0	20.76	1.0
	2k	22.06	2.4	78.08	3.8
	4k	54.14	6.0	182.56	8.8
	8k	101.10	11.0	342.64	16.0
Selector pattern	pseudo selector	6.26	1.0	27.83	1.0
	id selector	28.26	4.5	96.09	3.5
	class selector	35.49	5.7	115.77	4.2
	descendant selector	50.14	8.0	166.56	6.0
Property	color	78.04	1.0	271.23	1.0
	display	93.05	1.2	323.76	1.2
	width	242.22	3.1	843.07	3.1
	float	272.68	3.5	938.58	3.5

Table 2: CSS Microbenchmarking Results

of CSS selectors is an important metric because theoretically each CSS selector must be matched with each HTML tag to determine the style information of the webpage, leading to the computational complexity of O(#*tags* × #*rules*). We profile the number of total CSS selectors and show the results in Fig. 10. We observe a wide distribution of the selector count across webpages, ranging from 2,959 to 29.

To understand the impact of the number of selectors, we perform microbenchmarking. Each microbenchmark incrementally doubles the number of selectors from 1,000 to 8,000. As the "selector count" category in Tbl. 2 shows, the associated execution time and energy consumption scale dramatically increase as the number of selectors varies from 1,000 to 8,000. This suggests the selector count's significance.

In addition, the selection pattern of CSS selectors also has a strong impact on selector processing. For example, some patterns require traversing the DOM tree to identify the descendancy relationship between HTML tags, whereas others do not [11]. Fig. 10 shows the distribution of selection patterns. We observe hot patterns such as class and descendant.

We further conduct microbenchmarking on different selection patterns. Each microbenchmarking webpage repeats a particular pattern 1,000 times, and differs with a baseline that does not have CSS rules. The "selector pattern" category in Tbl. 2 reports the results. Depending on different patterns, the execution time and energy consumption differ. For example, a descendant pattern requiring DOM-tree traversing takes 8X processing time and consumes 6X energy compared to a pseudo pattern that does not involve DOM-tree operation.

Property In addition to selectors, we also profile CSS property usage. Fig. 11 shows the property distribution with the seven hottest properties and all others grouped as "others". The CSS property usage in webpages is largely diversified without any significantly hot properties. However, the number of total properties is vastly different across webpages.

Our microbenchmarking results of CSS properties (with each property repeated 1,000 times) is presented in the "property" category in Tbl. 2. We observe noticeable execution time and energy consumption differences between properties that

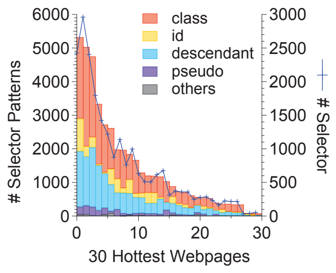


Fig. 10: CSS rule count and selector pattern distribution.

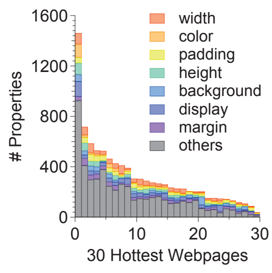


Fig. 11: CSS property distribution.

only affect individual HTML tags (such as color) and properties affecting other nodes in the DOM tree (such as width and float that affect positions of surrounding HTML tags).

6. Webpage Performance and Energy Modeling

The ability to predict webpage load time and energy consumption is vital to leverage big/little systems for intelligent scheduling. In this section, we demonstrate the feasibility of such predictions through regression modeling. Building on webpage characterizations described in the previous section, we apply regression modeling that captures different aspects of webpage characteristics and their interactions for predicting webpage load time and energy consumption (Sec. 6.1). Our models have a median prediction error rate of 5.7% and 6.4% for load time and energy consumption, respectively (Sec. 6.2).

6.1. Model Derivation

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

Linear Regression Modeling The linear regression model models the 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, $x = x_1, \ldots, x_p$ denote p predictors, and $\beta = \beta_0, \ldots, \beta_p$ 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) [33].

Predictors We first include the webpage primitives described in the previous section as model predictors because they show strong interwebpage differences, and as such have a strong influence on the load time and energy consumption.

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 3: Model Predictors

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 from webpage structures but which influence the load time and energy of rendering. We summarize these features in Tbl. 3. In total, we consider 376 predictors.

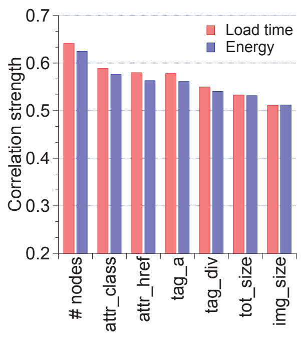
Obtaining Observations We require a number of sampling observations to construct the regression models. We obtain 2,500 sampling observations using the experimental setup described in Sec. 3. We measure both webpage load time and energy consumption simultaneously on the Cortex-A9 processor running at 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 [12] 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 [33]. 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. Fig. 12a shows the seven most-correlated predictors. For both load time and energy, we 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 Fig. 12b. 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



(a) Predictor-response correlation.

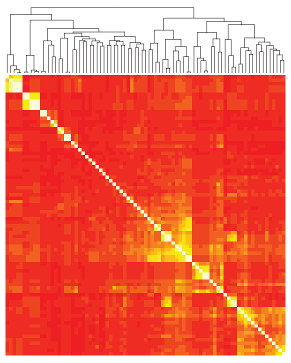
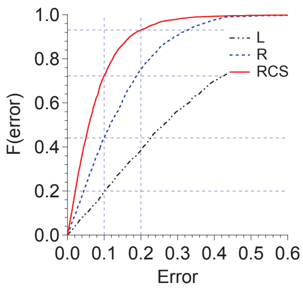
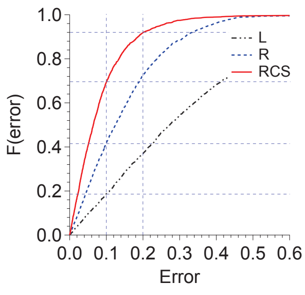

(b) Predictor self-correlation.

Fig. 12: Predictor correlations.



(a) Load time model.



(b) Energy model.

Fig. 13: CDF of prediction errors.

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 [33].

6.2. 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. Fig. 13a 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 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. Fig. 13b 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.

7. Dynamic Webpage Scheduling

Building on the prediction models described previously, we propose webpage-aware scheduling. The scheduler leverages the benefits of a big/little system by intelligently scheduling webpages for the ideal cut-off/energy trade-off (Sec. 7.1). 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 (Sec. 7.2).

7.1. Webpage-Aware Scheduling

Scheduler During the parsing stage, which takes <1% of the total execution time,[1] the webpage-aware scheduler extracts webpage characteristics, and feeds them into the prediction models to estimate the webpage load time and energy

[1] Based on our profiling. Prior work has also observed similar results [40].

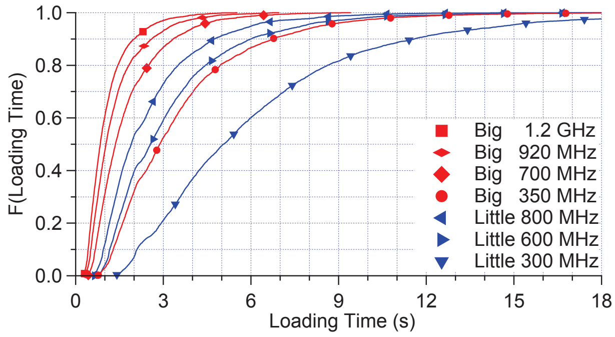
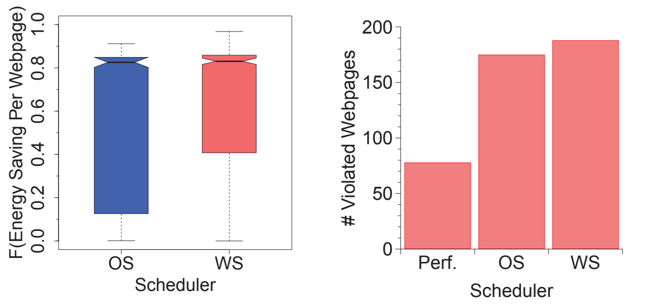


Fig. 14: CDF of webpage load time under different configurations.



(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. 15: Evaluation of different scheduling strategies.

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 [13]. For private cache warmup penalty, prior work shows that performance often improves when private LLCs of big and little cores are powered on together [30]. Thus, we ignore the warmup penalty. Also, prior work suggested that the power overhead of task migration is $<$ 0.75% [46]. 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.

7.2. Evaluation

Energy savings 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 [31, 49]. 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 heterogeneous scheduling across big/little cores, we synthesize such a scheduler by running the webpages under the "on-demand" cpufreq-governor [14] 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. Fig. 14 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. Therefore, we choose the big core (A9) with its peak frequency (1.2 GHz) as the high-performance baseline.

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, Fig. 15a 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.

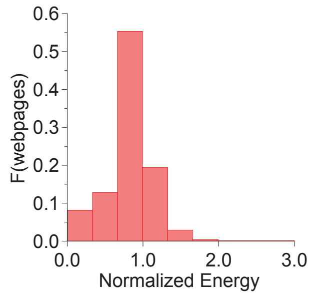
The webpage-aware scheduler has a denser energy-saving distribution toward 100% than the OS scheduler. This indicates that generally the webpage-aware scheduler achieves higher energy savings. Fig. 16a shows the histogram of per-webpage relative energy of the webpage-aware scheduler to the OS scheduler. The webpage-aware scheduler saves energy for about 80% of the webpages. There are several webpages that are misscheduled 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 (>2*X* in Fig. 16a). 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 Fig. 15b. 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.

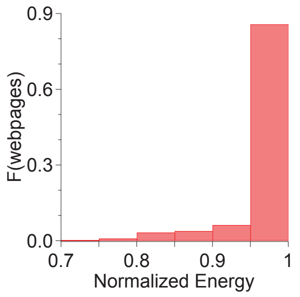
Cut-off sensitivity To assess the webpage-aware scheduler under variable user demands and mobile device conditions, we also experiment with different cut-off latencies. For example, when the end user requests faster webpage load at 2 seconds, the mechanism achieves 7.3% energy savings over the OS scheduler 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: overprediction causes webpages to load on a more powerful configuration that consumes more energy than the ideal one but does not cause cut-off violation; underprediction loads webpages on a weaker configuration that consumes less energy but violates the cut-off constraint. Our models lead to 10% overprediction and 4.1% underprediction. 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



(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. 16: Distribution of per-webpage energy comparisons.

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 Fig. 5) 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 Fig. 5) 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.

Discussion 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 [14]. For example, for `www.autoblog.com` (top-left in Fig. 5), 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.

Fig. 16b 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.

8. Related Work

JavaScript JavaScript is important in webpages. However, we believe it is a closely related, yet separate and distinct problem from webpage load. JavaScript code is most often executed after the webpage is loaded. Google recommends that, to improve performance (especially in mobile browsing), all JavaScript processing should be deferred until the webpage load finishes [15]. Proposals that speculatively process JavaScript [16, 38, 43] further separate JavaScript from webpage load. In addition, as pointed out in [47], a large portion of JavaScript code is either computationally intensive [17, 18] or an intensive stress on the memory management system. They require optimizations on the Just-In-Time (JIT) engine, programming model, or the garbage collector (GC) [28, 34, 39], all of which are beyond the scope of our work. Thus, our focus and mechanism in this paper are largely independent of JavaScript execution and its browser engine performance.

Web Browser Performance Optimization Current research proposals focus on parallelizing browser-specific tasks, such as parsing, CSS selection, etc. [24, 36, 40, 41]. 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/threads, leading to unacceptable energy inefficiency. Thus, we believe that while the parallelization methodology has potential in desktop computing, it is less favorable for mobile web browsing.

Another portion of web performance optimization focuses on improving the execution model of the web browser through asynchronous/multiprocess rendering, resource prefetching, smarter browser caching, etc. [19, 20, 37, 38, 50]. We focus on per-webpage processing, which can be incorporated with the improvement of the overall web browser architecture.

Web Browser Energy/Power Optimization Thiagarajan et al. [48] 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 [26, 29, 44], whereas we focus on improving the energy-efficiency of the mobile processor in response to the demand for high-performance. SiChrome [25] maps the critical executions of the browser into hardware to improve EDP, and is orthogonal to our work.

Single ISA Heterogeneous Scheduling Our webpage scheduling technique, based on big/little cores with the aid of DVFS, is an example of utilizing single-ISA heterogeneous systems for optimizing the performance versus energy tradeoff [35]. Nvidia's Kal-El [21] is a single-ISA heterogeneous system that integrates four high-frequency cores with one low-frequency core. ARM's proposed big.LITTLE system [13] contains a high-performance Cortex-A15 processor and a lower-performance but extremely energy-efficient Cortex-A7 processor. Without the availability of the big.LITTLE system, in this paper we use the Cortex-A9 and A8 to synthesize such a system, and show its advantage in high-performance and energy-efficient mobile web browsing. Our prediction-based scheduling technique is similar to other recent heterogeneous scheduling proposals, such as PIE [30]. However, instead of relying on (micro)architecture- and system-level statistics for prediction, we capture the complex behavior of webpage characteristics using regression modeling, and accurately predict the webpage load time and energy consumption.

Web Browser Workload Characterization Gutierrez et al. [32] perform microarchitecture-level characterization of the Android browser using 11 websites. We treat webpages, rather than the web browser, as the workload. Moreover, we mine 5,000 hot webpages to identify and quantify the inherent variance in webpages more rigorously, and correlate the webpage variance to the difference in webpage load time and energy. Butkiewicz et al. [27] take a similar approach, characterizing the complexity of different webpages. They construct a model for webpage load time considering server and network effects. Instead, we consider both load time and energy usage on the client side of mobile web browsing, and we perform webpage-aware scheduling on a heterogeneous system using even more fine-grained prediction models.

9. Discussion

We see a wide range of future applications for our webpage-aware approach. First, web developers can use our predictive models to reduce the rate of webpage abandonment by improving their webpage load time. Google provides webpage performance optimization techniques and microbenchmarking webpages to evaluate the different optimization techniques [22, 23]. Complementary to that, web developers can further rely on our predictive models to gather quantitative and fine-grained optimization results before deployment.

Second, our work is the first step in the process of integrating prediction and scheduling techniques into a mobile device with significant room for improvement. We can readily extend our scheme for system-level predictions with a more comprehensive and holistic understanding of the interactions among the different web browser components, such as the networking module, browser cache/database, etc. Including the features of HTML5 is also a topic for further improvement.

Finally, our approach is a heuristic for feedback-directed optimization. Future mobile web browsers can continuously optimize their execution based on the unique characteristics of a webpage or even a web application and their user requirements. We envision that such a browser will behave like a traditional runtime system that continuously coordinates its hardware resources for a specific optimization objective, not only limited to energy efficiency that this paper demonstrates.

10. Conclusion

We propose webpage-aware scheduling for high-performance and energy-efficient mobile web browsing. It is a new mechanism that dynamically matches the underlying heterogeneous hardware resources to webpages with diversified characteristics. Through detailed characterization of webpage variance, we apply regression modeling to predict webpage load time and energy consumption. Such predictive models allow the scheduler to identify the ideal core and frequency configurations for webpages to minimize the energy consumption under latency cut-off constraints. Real hardware and software measurements show that, on average, webpage-aware scheduling achieves 83.0% energy savings, while only violating the cut-off latency for 4.1% more webpages compared with a performance-oriented hardware strategy. Compared with a production-level OS DVFS scheduler, it achieves 8.6% energy savings and 4.0% performance improvement.

References

[1] KPCB 2012 Internet Trends. http://goo.gl/aXbVs

[2] KPCB 2012 Top Mobile Internet Trends. http://goo.gl/p8zU1

[3] Wayback Machine Internet Archive. http://archive.org/web/web.php

[4] CA Technologies: "It's offcial: web stress is bad for business". http://goo.gl/G2hwU

[5] Kissmetrics: "How loading time affects your bottom line". http://goo.gl/kosva

[6] Google: "The Google gospel of speed". http://goo.gl/hsZnF

[7] World Internet Usage Statistics News and World Population Stats. http://goo.gl/L8hG

[8] Pandaboard ES Rev B1 Schematic. http://goo.gl/biFQX

[9] RD2: "The three second rule". http://goo.gl/pynBl

[10] Google: "Web mertics: size and number of resources". http://goo.gl/FWdNp

[11] W3C: CSS Selectors. http://www.w3.org/TR/CSS2/selector.html

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

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

[14] Linux CPUFreq Governor. http://goo.gl/MzpYR

[15] Google: "Defer parsing of JavaScript". http://goo.gl/csWrR

[16] Web Workers. http://goo.gl/nejxO

[17] SunSpider JavaScript benchmark. http://goo.gl/Z52lw

[18] V8 Benchmark Suite - version 7. http://goo.gl/mALWo

[19] WebKit2. http://trac.webkit.org/wiki/WebKit2

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

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

[22] Google: "Web performance best practices". http://goo.gl/WlOnF

[23] Google: "Tutorials: make the web faster". http://goo.gl/CLHY0

[24] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum, "Towards parallelizing the layout engine of firefox," in *Proc. of USENIX HotPar*, 2010.

[25] V. Bhatt, N. Goulding-Hotta, Q. Zheng, J. Sampson, S. Swanson, and M. B. Taylor, "Sichrome: Mobile web browsing in hardware to save energy," *DaSi: First Dark Silicon Workshop*, 2012.

[26] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "The model is not enough: understanding energy consumption in mobile devices," in *Poster session of HotChip*, 2012.

[27] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Understanding website complexity: measurements, metrics, and implications," in *Proc. of IMC*, 2011.

[28] T. Cao, T. Gao, S. M. Blackburn, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proc. of ISCA*, 2012.

[29] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. of USENIX ATC*, 2010.

[30] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proc. of ISCA*, 2012.

[31] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas, "Policies for dynamic clock scheduling," in *Proc. of OSDI*, 2000.

[32] 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.

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

[34] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram, "Parallel programming for the web," in *Proc. of USENIX HotPar*, 2012.

[35] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. of MICRO*, 2003.

[36] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *Proc. of HPCA*, 2012.

[37] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "Pocketweb: instant web browsing for mobile devices," in *Proc. of ASPLOS*, 2012.

[38] H. Mai, S. Tang, S. T. King, C. Cascaval, and M. Pablo, "A case for parallelizing web pages," in *Proc. of USENIX HotPar*, 2012.

[39] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *Proc.of HPCA*, 2011.

[40] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proc. of WWW*, 2010.

[41] ——, "Ftl: Synthesizing a parallel layout engine," in *Eucopean Conference on Object-Oriented Program in Conjunction with PLDI*, 2012.

[42] L. A. Meyerovich, T. Mytkowicz, and W. Schulte, "Data parallel programming for irregular tree computations," in *Proc. of HotPar*, 2011.

[43] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster web browsing using speculative execution," in *Proc. of NSDI*, 2010.

[44] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. of EuroSys*, 2012.

[45] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proc. of PLDI*, 2011.

[46] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proc. of ISCA*, 2009.

[47] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Characterizing real-world behavior of javascript programs," in *Technical Report MSR-TR-2009-173, Microsoft Research*, 2009.

[48] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who killed my battery?: analyzing mobile browser energy consumption," in *Proc. of WWW*, 2012.

[49] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Proc. of OSDI*, 1994.

[50] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proc. of WWW*, 2010.