GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing

Yuhao Zhu Vijay Janapa Reddi

Department of Electrical and Computer Engineering, The University of Texas at Austin

yzhu@utexas.edu, vj@ece.utexas.edu

http://www.wattwiseweb.org/

Abstract

Web computing is gradually shifting toward mobile devices, in which the energy budget is severely constrained. As a result, Web developers must be conscious of energy efficiency. However, current Web languages provide developers little control over energy consumption. In this paper, we take a first step toward language-level research to enable energy-efficient Web computing. Our key motivation is that mobile systems can wisely budget energy usage if informed with user quality-of-service (QoS) constraints. To do this, programmers need new abstractions. We propose two language abstractions, QoS type and QoS target, to capture two fundamental aspects of user QoS experience. We then present `GreenWeb`, a set of language extensions that empower developers to easily express the QoS abstractions as program annotations. As a proof of concept, we develop a `GreenWeb` runtime, which intelligently determines how to deliver specified user QoS expectation while minimizing energy consumption. Overall, `GreenWeb` shows significant energy savings (29.2% $\sim$ 66.0%) over Android's default `Interactive` governor with few QoS violations. Our work demonstrates a promising first step toward language innovations for energy-efficient Web computing.

Categories and Subject Descriptors D.3.2 [*Programming Language*]: Language Classifications–Specialized application languages; D.3.3 [*Programming Language*]: Language Constructs and Features–Constraints

Keywords Energy-efficiency, Web, Mobile computing

1. Introduction

Web computing is gradually shifting toward mobile devices. As of 2014, mobile devices already generate more Internet traffic than desktops, and the gap is exponentially growing [22]. The major challenge for mobile Web computing is the mobile devices' tight battery budget, which severely limits sustained operation time and leads to user frustration. Recent statistics shows that poor energy behavior is a top reason that causes negative App reviews [1], and 55% of mobile users would delete an App that exhibits heavy battery usage [27]. Mobile users are now aware of major energy consumers through energy monitoring and diagnosis Apps. For example, both Android and iOS provide built-in functionality showing each application's battery usage.

Because of the increasing user awareness, Web developers today must be 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 [57, 71, 73]. 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 [17].

Language Support In this paper, we present `GreenWeb`, a set of Web language extensions defined as Cascading Style Sheet (CSS) rules that allow Web developers to express user QoS expectations at an abstract level. Based on programmer-guided QoS information, the runtime substrate of `GreenWeb` dynamically determines how to deliver the target QoS experience while minimizing the energy consumption.

To help Web developers easily express QoS information 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 in-

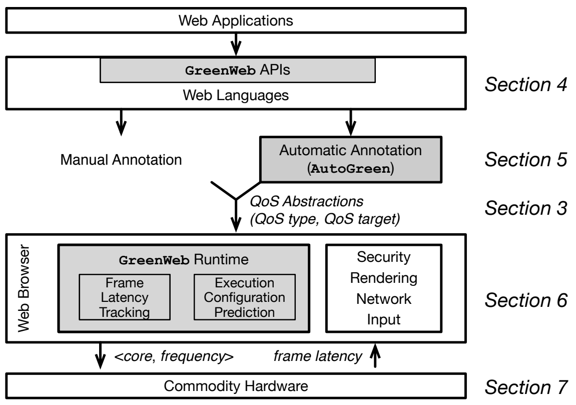


Fig. 1: `GreenWeb` system overview. Enhancements to the existing Web stack are shaded. Web applications annotated with `GreenWeb` APIs provide QoS information to the `GreenWeb` runtime. Based on the QoS information, the runtime exploits energy-delay trade-off in commodity hardware to minimize energy while meeting Qos expectations. The hardware also continuously provides feedback to the `GreenWeb` runtime to enable adaptive optimizations.

teraction 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 exact knowledge of code logic. They can provide critical 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 the iterations in a `for` loop are completely independent so that the runtime can safely parallelize the loop without the need to check for correctness. Similarly, `GreenWeb` annotations allow the Web runtime to perform "best-effort" energy optimizations while still guaranteeing a satisfactory QoS experience without having to infer QoS information.

Design and Implementation Fig. 1 provides an overview of the system. Web applications are annotated with `GreenWeb` APIs to provide QoS information (i.e., QoS type and QoS target) to the `GreenWeb` runtime. The `GreenWeb` runtime is designed as a new Web browser component sitting alongside existing modules. Annotation is conducted either manually by developers or through an automatic annotation framework we develop called AUTOGREEN.

While the `GreenWeb` language extensions do not mandate any specific runtime implementation and can support any optimization strategy for QoS-aware energy efficiency in general, we specifically focus on leveraging the asymmetric chip-multiprocessor (ACMP) hardware architecture [54, 68]. ACMP is long known to provide a wide performance-energy trade-off space, and is already widely used in today's mobile systems [6, 16]. Based on the QoS information, the `GreenWeb` runtime predicts the ideal ACMP configuration that minimizes energy consumption while meeting specified QoS expectations. The runtime also continuously monitors hardware execution to enable adaptive optimizations.

We implement `GreenWeb` in the open-source Chromium browser engine [7], which is directly used in Google's Chrome browser and is the core for many other popular Web browsers such as Opera and the Android default browser. We evaluate `GreenWeb` on the ODroid XU+E development board [24], which contains an Exynos 5410 System-on-chip (SoC) that is used in the Samsung Galaxy S4 smartphone. Our results show that applications with `GreenWeb` annotations achieve 29.2% $\sim$ 66.0% energy savings with only small QoS violations comparing to Android's `interactive` governor, which is a common energy optimization strategy for interactive use.

Contributions We make four contributions in the paper:

- We identify two programming abstractions of user QoS experience, QoS type and QoS target, that are critical to QoS-aware energy efficiency optimizations.
- We present `GreenWeb`, a set of language extensions that allow developers to express the two QoS abstractions and guide energy optimizations in mobile Web applications.
- We demonstrate one candidate design of a `GreenWeb` runtime, which leverages the ACMP heterogeneous CPU architecture that is already prevalent in today's mobile hardware and achieves significant energy savings over current energy optimization strategies.
- We present AUTOGREEN, an automatic annotation framework that improves developer productivity by applying `GreenWeb` annotations without developers intervention.

We make our `GreenWeb` language specification, design, and implementation as well as the AUTOGREEN framework publicly available at: `http://wattwiseweb.org/`.

The rest of this paper is organized as follows. Sec. 2 introduces the background and describes the scope of the Web that this paper discusses. Sec. 3 defines two abstractions that are critical to mobile user QoS experience, and Sec. 4 describes the proposed `GreenWeb` language constructs that express the two abstractions. Sec. 5 presents AUTOGREEN to demonstrate the feasibility of automatically applying `GreenWeb` annotations to a Web application. Sec. 6 presents a particular `GreenWeb` runtime design that leverages the ACMP architecture. Sec. 7 quantifies the benefits of a `GreenWeb`-enabled Web runtime. Sec. 8 discusses the implications and limitations of the current design and implementation of `GreenWeb`. Sec. 9 puts `GreenWeb` in the broad context of related work, and Sec. 10 concludes the paper.

2. Web: A Universal Application Platform

In this section, we first present the broad scope of the Web applications we discuss in this paper. We then briefly introduce Web languages and the Web browser runtime. Overall, we 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 runtime design. These observations motivate our effort on `GreenWeb` that extends existing Web languages with new semantics to enable energy efficiency optimizations.

Web Applications 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 [4]. Strategy Analytics reported that by the year 2015 63% of all business mobile applications are based on Web technologies [18].

Mobile system vendors are actively embracing Web technologies. Both iOS and Android provide developers APIs that expose Web browser functionalities [5, 20]. 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 [34]. In this paper, the scope of Web application extends beyond webpages to also include such hybrid applications.

Web Languages and 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 [15], 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. Specifically, a Web browser typically consists of a rendering engine that translates HTML and CSS, and a JavaScript engine that executes JavaScript code.

Over the past two decades, language evolution and Web runtime design have been constantly driving Web innovations [28, 29]. 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.

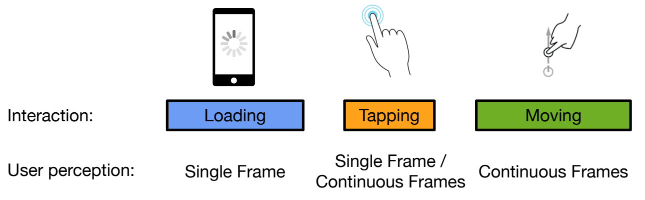


Fig. 2: The LTM (Loading-Tapping-Moving) user-application interaction model of mobile Web. LTM captures three primitive types of interaction: page loading, finger tapping, and finger moving. We use LTM as a framework to reason about user QoS experience.

3. QoS Abstractions for the Mobile Web

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 (Sec. 4) describes our 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 introduce a user-application interaction model called LTM. LTM captures three fundamental user interactions in mobile Web applications (Loading-Tapping-Moving) and gives us a framework for reasoning about user QoS experience (Sec. 3.1). Based on the LTM model, we propose the QoS type (Sec. 3.2) and QoS target (Sec. 3.3) abstractions. We discuss why they are necessary and sufficient to express QoS information for QoS-aware energy efficiency optimizations.

3.1 LTM Model of Mobile User Interaction

QoS experience is fundamentally linked with user-application interactions. 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). Fig. 2 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). Specifically, the moving interaction 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.

Each event is bound to a DOM node with a callback function, which is executed when the event is triggered on the associated DOM node. The result of callback execution is fed into the Web browser rendering engine, which eventually paints the resulting frame(s) and updates the display. Frames are what users perceive as application's responses to their interactions, and thus determine the QoS experience. In the following subsections, we propose two abstractions for QoS experience based on different frame characteristics.

3.2 QoS Type Abstraction

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 [45]. 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 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" [30], 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. Determining the exact sequence of frames given an input event is a non-trivial task and we discuss it in Sec. 6.4.

Table 1: Interactions in mobile Web applications fall into three categories based on different QoS type and QoS target combinations.

QoS Type	QoS Target (T_I, T_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

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 [31] and CSS animation/transition [9, 12].

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 Sec. 4 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.

3.3 QoS Target Abstraction

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 (T_I) and usable target (T_U) [71]. Imperceptible target delivers a latency that is im-

```
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. 3: The syntax of `GreenWeb` language extensions.

perceptible/instantaneous to users. Achieving a performance higher than T_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 T_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 [45]. 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 [26]. Thus, 100 ms and 300 ms can be used as the T_I and T_U values, respectively.

In contrast, when users are aware of a computationally intensive job being processed, they tend to have high tolerance for latencies [61]. 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 [39, 58]. Therefore, 1 second and 10 seconds can be treated as the T_I and T_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 [42]. 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 [21, 23].

User interactions fall into three distinct categories based on the different QoS type and QoS target combinations as listed in Table 1. Although the absolute values of QoS target (T_I and T_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.

Table 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 T_I and T_U values in Table 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 T_I and T_U values in Table 1 as the default QoS target.
`E:QoS { onevent-qos: continuous | single, ti-value, tu-value }`	Explicitly specify T_I (`ti-value`) and T_U values (`tu-value`) for QoS targets. Note that both values must either appear or be ommitted together.

4. GreenWeb Language 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 describe the design and specification of `GreenWeb` (Sec. 4.1). We then present usage scenarios to demonstrate the expressiveness and modularity of the `GreenWeb` design (Sec. 4.2).

4.1 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 richer information such as controlling animations [9] and adapting to different device form factors [14]. `GreenWeb` follows this spirit of CSS language evolution and further expands the CSS semantics scope to allow expressing user QoS related information.

Fig. 3 shows the `GreenWeb` syntax, and Table 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

```
 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. 4: Express the QoS type of `ontouchstart` event as “continuous,” and use the default T_I and T_U values.

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

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 [10] “`: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 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 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 T_I and T_U values in Table 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 T_I and T_U (in milliseconds) after `single` or `continuous`, as shown in the third rule in Table 2.

With the knowledge of new language constructs that the `GreenWeb` introduces, we now examine a few common usage scenarios of the `GreenWeb` APIs.

4.2 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 [12]. Fig. 4 shows an example in which the transition of the `width` property of a `div` element is animated. The initial `width` property is set to 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 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. Fig. 5 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 [33] arrives) [21] 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 developers add QoS annotations for an event independent of how the event callback is implemented. In other words, the `GreenWeb` let QoS and functionality of Web programming be two separate concerns.

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.

5. 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.

Fig. 6 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, 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 [8, 11], 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.

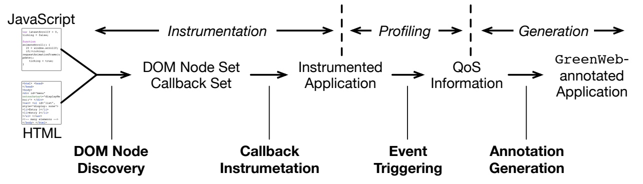


Fig. 6: AUTOGREEN's workflow to automatically annotate mobile Web applications with `GreenWeb` APIs.

AUTOGREEN uses the default QoS target values listed in Table 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. GreenWeb Runtime Design

While `GreenWeb` APIs specify *what* QoS type and QoS target that users expect, it is the job of the runtime substrate to determine *how* user QoS expectations are satisfied in an energy-efficient manner. We first state the problem that `GreenWeb` runtime addresses (Sec. 6.1). We then describe the general scheme that `GreenWeb` runtime uses (Sec. 6.2). Finally, we discuss two critical components of the runtime design (Sec. 6.3 and Sec. 6.4).

6.1 Problem Statement

In this paper, we choose to explore the runtime design by exploiting the ACMP hardware architecture [54, 68]. Alternatively, a runtime system that performs optimizations purely at the software level (e.g., selective resource loading [38]) or at the SoC-level (e.g., use power-conserving colors [44] or dynamic display resolution scaling [50]) is also possible.

The ACMP architecture consists of multiple cores with different microarchitectures, such as out-of-order and in-order. Each core has a variety of frequency settings. Different core and frequency combinations thus provide a large trade-off space between performance and energy. The objective of our ACMP-based `GreenWeb` runtime is to find an ideal ACMP execution configuration (i.e., a $\langle core, frequency \rangle$ tuple) such that the QoS target is satisfied with minimal energy.

It is important to emphasize that the `GreenWeb` runtime operates on a per-frame basis because frames are what ultimately dictate user perceivable experience as discussed in Sec. 3.1. That is, if an event's QoS type is "single," the runtime finds the ideal execution configuration for the single frame associated with the event. If an event's QoS type is "continuous," 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 as enforced by the `GreenWeb` API semantics shown in Table 2.

Given the problem statement, we now describe the general scheme that the `GreenWeb` runtime uses in Sec. 6.2. We then discuss two critical components of the runtime design. Specifically, Sec. 6.3 discusses frame latency tracking that enables optimization for any *single* frame, based on which Sec. 6.4 discusses how to determine the associated frames of an event, which allows optimizing for *all* the frames.

6.2 Execution Configuration Prediction

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

The energy model can be built based on the performance model and the power consumption under different core and frequency settings. We profile the different power consumptions statically and hard-code them into the runtime. In this section, we explain the performance model.

We base the performance model on the classical DVFS analytical model initially proposed by Xie et al. [70] and used in many subsequent works [57, 69, 71]:

$$T = T_{independent} + N_{nonoverlap}/f \qquad (1)$$

in which T is the frame latency; f is the CPU frequency; $T_{independent}$ is the time that is independent of f, which primarily includes the GPU processing and main memory access time; $N_{nonoverlap}$ is the number of CPU cycles that do not overlap with $T_{independent}$ and scales with f.

Equ. 1 is a system of equations with two unknown variables $T_{independent}$ and $N_{nonoverlap}$. We solve Equ. 1 by profiling frame latencies twice, one at the maximum frequency and one at the minimum frequency. Measuring frame latency is non-trivial. It is performed by the *latency tracking* runtime module, which we will discuss in Sec. 6.3. We build performance models for big and little cores separately because different microarchitectures inherently have different $T_{independent}$ and $N_{nonoverlap}$ characteristics.

The `GreenWeb` runtime uses measured frame latencies as feedback information to fine-tune the prediction, similar to the event-based scheduling [71]. If the previous prediction leads to a frame latency that violates the QoS target (i.e., under-predicts), the `GreenWeb` runtime increases the frequency to the next available level or transitions the execution from the little core to the big core. The opposite adjustment is applied when the model over-predicts. If the model mispredicts consecutively more than a certain threshold, the runtime initiates new profilings to recalibrate the model.

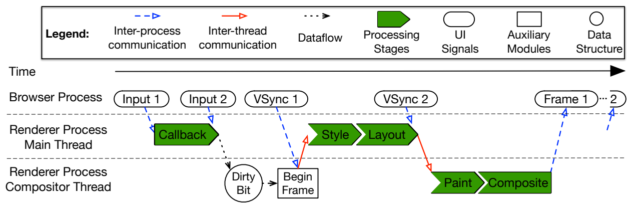


Fig. 7: The simplified view of frame lifetime in modern multi-process/thread browsers. A frame starts when the browser process receives an input event and ends when the frame is displayed and the browser process is signaled. In between, an input event is processed by different stages spread across multiple threads. Different input events might interleave with each other.

6.3 Frame Latency Tracking

Tracking frame latency is crucial to constructing 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. To the best of our knowledge, this is the first work that describes tracking frame latencies in Web applications. Most prior work either is concerned only with the callback latency [40, 71], 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 [64].

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, Fig. 7 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 [25]. 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 Equ. 1 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 Fig. 7, Input 2 is triggered before Input 1 finishes. Naively associating an input event with its immediate next frame in

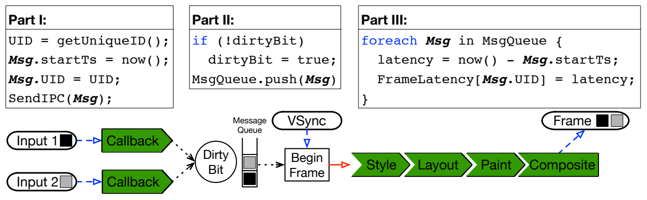

Part I:
```
UID = getUniqueID();
Msg.startTs = now();
Msg.UID = UID;
SendIPC(Msg);
```
Part II:
```
if (!dirtyBit)
  dirtyBit = true;
MsgQueue.push(Msg)
```
Part III:
```
foreach Msg in MsgQueue {
  latency = now() - Msg.startTs;
  FrameLatency[Msg.UID] = latency;
}
```

Fig. 8: 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.

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 [21, 33]. 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 callback 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 Fig. 8. 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).

6.4 Associating Frames with Events

The `GreenWeb` runtime operates on a per-frame basis. That is, after an event is triggered the runtime predicts the ideal ACMP configuration of each frame until all the frames associated with the event are produced. Therefore, it is important to determine all the frames associated with an event as all the associated frames share the same QoS target of the event.

Intuitively, an event's associated frames refer to all the frames that are generated because of the event being triggered. Associating frames with events must be performed at runtime because frames are dynamically generated as an event is being processed. It is difficult (and theoretically impossible) for the Web runtime to determine a priori the associated frames immediately after an event is triggered. The `GreenWeb` runtime leverages an event's QoS type information to calculate its associated frames on-the-fly as follows.

If an event's QoS type is "single," there is only one frame associated with the event, and that is the frame directly produced by the triggering event. In this case, the associated frame can be determined by simply following the frame latency tracking algorithm as discussed in Sec. 6.3.

If an event's QoS type is "continuous," calculating its associated frames is equivalent to forming a transitive closure of all the frames from the root event. Recall from Sec. 6.3 that a frame goes through a set of processing stages spread across different processes and threads before it is generated. The runtime starts from the root event and follows all the inter-process messages (e.g., `IPC::Message` in Chrome) and inter-thread messages (e.g., `PostTask` in Chrome) generated by the root event. The runtime adds all the frames reacheable from the root event until no message is generated, at which time all the associated frames are found. The found frames get the same QoS target of the event, based on which the runtime predicts the ACMP configuration.

7. Evaluation

We first introduce our experimental setup (Sec. 7.1). We then use a set of microbenchmarks to understand the effectiveness of `GreenWeb` on *invididual* events that have different QoS characteristics (Sec. 7.2). We present results on full interactions consisting of a sequence of events in the end (Sec. 7.3).

7.1 Experimental Setup

Software Infrastructure We implement `GreenWeb` and its runtime system in 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. Our implementation is based on Chromium version 48.0.2549.0, which is the most recent version at the time of our work. The modified Chromium runs on unmodified Android version 4.2.2.

Hardware Platform We use the ODroid XU+E development board [24], 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 [71, 73].

Energy Measurement `GreenWeb` 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 [48].

We measure the processor power and energy consumption on real hardware 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 (*not* the estimated time from the timing prediction model).

Application Selection Table 3 shows the applications we use for evaluation. We crawl them using HTTrack [19] 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 [51, 72, 73]. 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 [2].

Baseline We compare `GreenWeb` 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 [66] shows that mobile users often have to interact with devices under different battery conditions. Therefore, we evaluate `GreenWeb` 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. It corresponds to the imperceptible QoS experience discussed in Sec. 3.3. The imperceptible performance threshold T_I is used as the QoS target.
- "Usable" represents scenarios in which the battery budget is tight and users could tolerate lower performance. It corresponds to the usable QoS experience. The usable performance threshold T_U is used as the QoS target.

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

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 [47], 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.

Table 3: List of applications. "Time" indicates full interaction duration. "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 Sec. 3.1. Applications marked with * are manually annotated because they are developed using libraries that AutoGreen does not currently support. Their annotation percentage numbers are estimated.

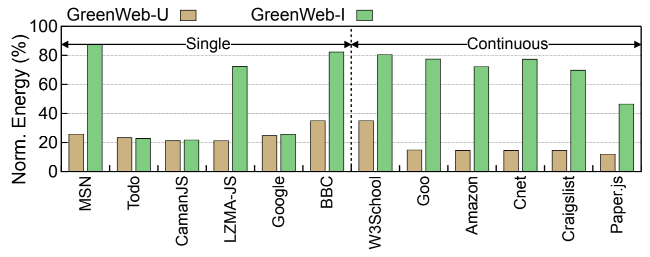
	Micro-benchmarking			Full Interaction		
Application	**Interaction**	**QoS Type**	**QoS Target**	**Time**	**Events**	**Annotation**
BBC	Loading	Single	(1, 10) s	0:86	60	20%*
Google	Loading	Single	(1, 10) s	0:31	26	87.5%
CamanJS	Tapping	Single	(1, 10) s	0:49	24	100%
LZMA-JS	Tapping	Single	(1, 10) s	0:53	39	100%
MSN	Tapping	Single	(100, 300) ms	0:59	126	51.2%
Todo	Tapping	Single	(100, 300) ms	0:26	26	38.3%
Amazon	Moving	Continuous	(16.6, 33.3) ms	0:36	101	33%*
Craigslist	Moving	Continuous	(16.6, 33.3) ms	0:25	22	84.6%
Paper.js	Moving	Continuous	(16.6, 33.3) ms	0:16	560	100%
Cnet	Tapping	Continuous	(16.6, 33.3) ms	0:46	60	55.3%
Goo.ne.jp	Tapping	Continuous	(16.6, 33.3) ms	0:16	23	51.8%
W3Schools	Tapping	Continuous	(16.6, 33.3) ms	0:64	59	100%

7.2 Microbenchmarking Results

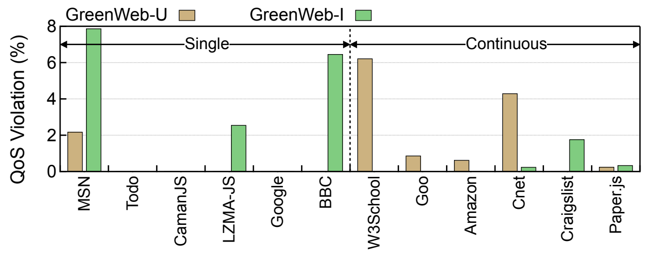
To understand the effectiveness of `GreenWeb` on *individual* events, we design a set of microbenchmarking experiments. The goal is to exercise `GreenWeb` on various events that differ in interaction types (LTM), QoS type, and QoS target. To better understand the behavior of `GreenWeb` during microbenchmarking,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

We construct the microbenchmark set by pairing each application 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 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 Fig. 9a 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.

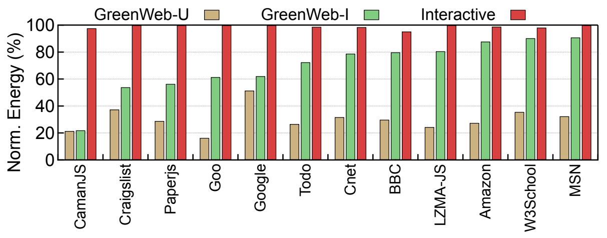


(a) Energy consumption normalized to *Perf*. Lower is better.

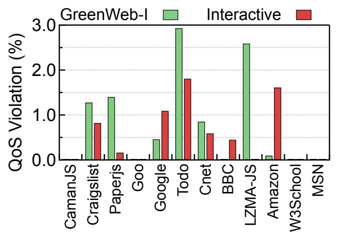


(b) QoS violations normalized to *Perf*. Lower is better.

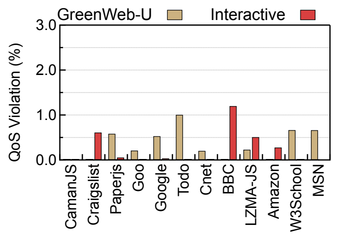
Fig. 9: Microbenchmarking results. Energy numbers are normalized to *Perf*, which provides the best QoS and consumes the most energy. QoS violations are presented as additional violations on top of *Perf*. *GreenWeb-I* and *GreenWeb-U* are `GreenWeb` under two usage scenarios.



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



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



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

Fig. 10: Full interaction results. Energy savings are normalized to *Perf*. QoS violations are presented as additional violations on top of *Perf*.

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).

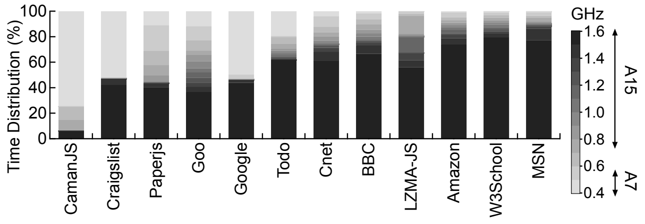
QoS Violation QoS violation is defined as the percentage by which a frame latency exceeds the QoS target. For example, a frame latency of 200 ms leads to an 100% QoS violation under a 100 ms QoS target. For events with a "continuous" QoS type, we report the geometric mean of all associated frames. It 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.

Fig. 9b 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 Sec. 6.2). 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.

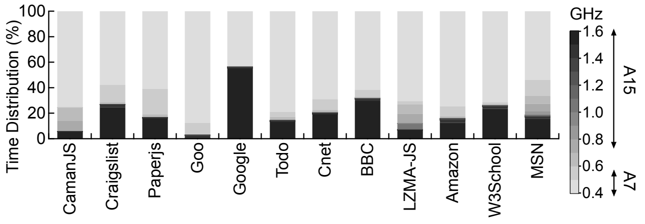
Some events that have a "continuous" QoS type have relatively high QoS violations under the usable mode. Outstanding examples are W3School and Cnet. Our analysis shows that most of the QoS violations come from frame complexity surges in a continuous frame sequence. `GreenWeb` aggressively scales down 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 [57].

7.3 Full Interaction Evaluation

In this section, we perform a sequence of interactions on each application, and evaluate the end-to-end behavior of `GreenWeb`. Each sequence consists of a mix of LTM interactions and contains events with different QoS types and QoS



(a) Architecture configuration distribution for *GreenWeb-I*.



(b) Architecture configuration distribution for *GreenWeb-U*.

Fig. 11: The architecture configuration distribution under the "imperceptible" (*GreenWeb-I*) and "usable" (*GreenWeb-U*) usage scenario.

targets. The "Full Interaction" category in Table 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.

Annotation Effort To annotate applications for full interaction evaluation, we use a combination of AUTOGREEN and manual annotation. We use AUTOGREEN because of the large amount of events in each application. Automatic annotation greatly improves productivity. However, AUTOGREEN does not always annotate QoS targets correctly because it conservatively assumes short response latency for events with a "single" QoS type (see Sec. 5). Therefore, we manually correct the QoS target for events that should have a long response latency. The "Annotation" column in Table 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 Sec. 3.1.

Overall, it took authors about 5 $\sim$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.

Energy Savings Fig. 10a shows the energy consumption of *Interactive* and `GreenWeb`'s two usage scenarios. The results are normalized to *Perf*, and sorted in the ascending order of *GreenWeb-I*. As compared to *Interactive*, `GreenWeb` 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 leads to high CPU utilization. *Interactive* responds to the high CPU utilization by increasing CPU performance. With the QoS knowledge provided by developers,

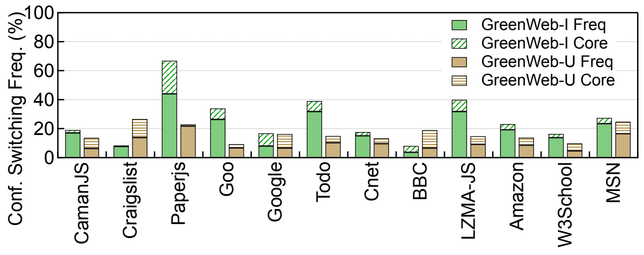


Fig. 12: Execution configuration switching frequency under *GreenWeb-I* and *GreenWeb-U*. Two configuration switching types exist: CPU frequency switch (solid) and core migrations (stripe).

however, `GreenWeb` can identify execution configurations that conserve energy while still meeting QoS requirements.

Architecture Configuration Distribution To better understand the sources of energy savings of `GreenWeb`, we examine the architecture configuration distribution of `GreenWeb` under the imperceptible and usable usage scenario shown in Fig. 11a and Fig. 11b, respectively. Bars with darker colors indicate higher performance configurations.

We make two notable observations from the distribution results. First, `GreenWeb` tends to bias toward big core (A15) configurations much more often under the imperceptible scenario (Fig. 11a) than under the usable scenario (Fig. 11b). This observation confirms the result that *GreenWeb-I* has less energy saving than *GreenWeb-U*. Second, the fact the `GreenWeb` dynamically changes its execution configuration under different QoS targets indicates that the `GreenWeb` 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, Fig. 12 shows the switching frequency of architecture configuration in *GreenWeb-I* and *GreenWeb-U*. We decompose the configuration switching into two categories: CPU frequency change and core migration (between big and little clusters). Thus, Fig. 12 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, `GreenWeb` introduces only modest configuration switching (20% on average). Recall from Sec. 7.1 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 ms. Therefore, the execution configuration has minimal performance impact.

Second, for most of applications *GreenWeb-I* incurs more switchings than *GreenWeb-U*. This is unsurprising because as compared to *GreenWeb-U*, *GreenWeb-I* optimizes for a tighter QoS target, which is more sensitive to frame (phase) variance and more vulnerable to frame performance misprediction. 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 [57] 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 [37, 53] is also beneficial in mobile CPUs.

QoS Violation Fig. 10b and Fig. 10c show the QoS violation of *Interactive* and `GreenWeb` under the imperceptible and usable scenarios, respectively. On average, `GreenWeb` 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*, `GreenWeb` has similar, in some cases fewer, QoS violations. Considering the significant energy savings, we conclude that the QoS-aware `GreenWeb` system can use energy more wisely by being aware of user QoS expectations. Overall, `GreenWeb` achieves better energy efficiency than the QoS-agnostic *Interactive* scheme.

8. Discussion

Manual Annotation vs. Runtime Mechanism As an alternative to receiving QoS annotations from developers, the Web runtime could try to detect QoS information at runtime without language hints. One strategy is to implement the exact logic of AUTOGREEN within the Web runtime. 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 in this paper 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) [53]. 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) [32] 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 [55]) 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.

9. Related Work

Language Support for Web Performance The Web community has a long tradition of providing language extensions that allow developers to specify "hints" for browsers. The focus, however, has been primarily on *performance* optimizations. `GreenWeb`, to the best of our knowledge, is the first Web language extension that specifically targets *energy*.

The most classical example of a performance hint is link prefetch [46], which lets Web developers use an HTML tag to specify that a particular link will likely be fetched shortly. 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 [13], 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.

Energy Optimizations in Mobile Systems There have been a few proposals on specific runtime systems that make trade-off between user QoS and energy for mobile (Web) applications. Zhu et al. leverage ACMP to improve the energy efficiency of webpage loading without violating cut-off deadlines [73]. Event-based scheduling (EBS) trades off event execution latency with energy consumption in mobile Web applications [71]. Nachiappan et al. proposed to coordinate different SoC components to improve the energy efficiency of animation-based mobile applications [59, 60].

`GreenWeb` fundamentally differs from these prior proposals because `GreenWeb` is not a particular runtime implementation; rather it is a language extension design that enables general QoS and energy trade-offs without posing constraints on specific runtime implementations. `GreenWeb` allows developers to express critical user QoS information, and it is up to the runtime designers to decide how to best deliver QoS in an energy-efficient manner.

In addition, with the programmer-assisted QoS hints from `GreenWeb` previous Web runtimes can be better guided to make a calculated trade-off between user QoS and energy. For example, without QoS annotations EBS relies on runtime measurement of event latency as a proxy for user QoS expectations. If an event takes a long time to execute, EBS "guesses" that it is an event for which users could naturally tolerate a long latency and, thus, decides to reduce CPU frequency. However, the measured latency is merely an artifact of a particular mobile system's capability such as its CPU performance. `GreenWeb` annotations express inherent user expectations and thus provide definitive QoS constraints.

Recent proposals on microarchitecture support for improving the efficiency of event-driven Web applications at the hardware level, such as ESP [41], EFetch [40], and WebCore [74], are complementary to our language-level work.

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 [62] or require developers to improve manually the energy efficiency [49, 56, 63]. `GreenWeb` eases developers' effort by automatically optimizing for energy efficiency.

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 [67] 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 [36] 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 [43] and EnerJ [65] 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 [52] 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.

10. Conclusion

As a promising first step, our work demonstrates language innovations to enable energy-efficient Web computing. The proposed `GreenWeb` language extensions effectively integrate application developers into the energy optimization loop by empowering them to express user QoS expectations at an abstract level. With such developer-assisted "hints," the runtime can achieve significant energy savings.

GreenWeb language extensions do not pose constraints on specific runtime implementations. Instead, GreenWeb provides a general way of making trade-offs between QoS and energy consumption. In this paper, we demonstrate one particular implementation that leverages the big/little ACMP architecture. It is also feasible to build a runtime leveraging only a single big (or little) core capable of DVFS [35, 57]. In addition, one could implement a GreenWeb runtime using pure software-level techniques or leveraging SoC-level knowledge. Having said this, the ACMP architecture is already widely adopted in today's mobile SoCs shipped by major vendors such as Samsung and Qualcomm [16]. We expect our GreenWeb implementation to be readily applicable on commodity mobile hardware.

Acknowledgments

We thank the Trinity research group members, the anonymous reviewers, and our shepherd Adrian Sampson for the many comments that have contributed to this work. Special thanks to Nat Duca and Vladimir Levin from the Google Chrome team for the helpful technical discussions. This work is partially supported by a Google research award. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] "9 Causes of Bad App Reviews." http://blog.monkop.com/post/120657007496/9-causes-of-bad-app-reviews
[2] "Alexa." http://www.alexa.com/
[3] "Android CPUFreq Governors." https://android.googlesource.com/kernel/common/+/android-4.4/Documentation/cpu-freq/governors.txt
[4] "Android Fragmentation Visualized." http://opensignal.com/assets/pdf/reports/2014_08_fragmentation_report.pdf
[5] "Android WebView APIs." http://developer.android.com/reference/android/webkit/WebView.html
[6] "big.LITTLE Technology: The Future of Mobile." https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf
[7] "Chromium browser." http://www.chromium.org/Home
[8] "CSS animation Event." https://developer.mozilla.org/en-US/docs/Web/Events/animationend
[9] "CSS Animations." http://www.w3.org/TR/css3-animations/
[10] "CSS Pseudo-classes." http://www.w3.org/TR/selectors/#pseudo-classes
[11] "CSS transitionend Event." https://developer.mozilla.org/en-US/docs/Web/Events/transitionend
[12] "CSS Transitions." http://www.w3.org/TR/css3-transitions/
[13] "CSS Will Change Module Level 1." http://www.w3.org/TR/css-will-change-1/
[14] "CSS3 Media Queries." http://www.w3.org/TR/css3-mediaqueries/
[15] "Document Object Model (DOM)." http://www.w3.org/DOM/
[16] "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology." https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf
[17] "How 1s Could Cost Amazon $1.6 Billion in Sales." http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales
[18] "HTML5 On The Rise: No Longer Ahead Of Its Time." http://techcrunch.com/2015/10/28/html5-on-the-rise-no-longer-ahead-of-its-time/
[19] "HTTrack." https://www.httrack.com/
[20] "iOS Developer Library: UIWebView." https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/
[21] "Jank Busting for Better Rendering Performance." http://www.html5rocks.com/en/tutorials/speed/rendering/
[22] "KPCB 2015 Internet Trends." http://www.kpcb.com/blog/2015-internet-trends
[23] "NVidia: Adaptive VSync Technology." http://www.geforce.com/hardware/technology/adaptive-vsync/technology
[24] "ODROID-XU+E Development Board." http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079
[25] "Rendering Performance." https://developers.google.com/web/fundamentals/performance/rendering/
[26] "Speed, Performance, and Human Perception." http://chimera.labs.oreilly.com/books/1230000000545/ch10.html#SPEED_PERFORMANCE_HUMAN_PERCEPTION
[27] "Survey: Exploring the Reasons Users Complain about Apps." http://www.fiercedeveloper.com/story/survey-exploring-reasons-users-complain-about-apps/2012-11-09
[28] "The Evolution of HTML5." http://www.instantshift.com/2012/07/20/the-evolution-of-html5-infographic/
[29] "The Evolution of the Web." http://www.evolutionoftheweb.com/
[30] "Time-to-first-X-paint Metrics: Status and Refinement Plans." https://docs.google.com/document/d/1Owfs6arciEnWgT2-8bWCcHdYRIKRKZ0Xj8UtqRx4c3k/edit
[31] "Timing Control for Script-based Animations." http://www.w3.org/TR/animation-timing/
[32] "User Agent Intervention." http://bit.ly/user-agent-intervention
[33] "V-sync." https://en.wikipedia.org/wiki/Screen_tearing#V-sync
[34] "Your Favourite App isnt Native." http://kennethormandy.com/journal/your-favourite-app-isnt-native
[35] "Nvidia Tegra 4 Family CPU Architecture: 4-PLUS-1 Quad core," in *Nvidia Whitepaper*, 2013.
[36] W. Baek and T. M. Chilimbi, "Green: a Framework for Supporting Energy-conscious Programming using Controlled Approximation," in *Proc. of PLDI*, 2010.
[37] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill, "FIVR: Fully Integrated Voltage Regulators on 4th Generation Intel Core SoCs," in *Proc. of APEC*, 2014.
[38] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing Web Content to Improve

User Experience on Mobile Devices," in *Proc. of NSDI*, 2015.

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

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

[41] ——, "Accelerating Asynchronous Programs through Event Sneak Peek," in *Proc. of ISCA*, 2015.

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

[43] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu, "Energy Types," in *Proc. of OOPSLA*, 2012.

[44] M. Dong and L. Zhong, "Chameleon: a Color-adaptive Web Browser for Mobile OLED Displays," in *Proc. of MobiSys*, 2012.

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

[46] D. Fisher and G. Saksena, "Link Prefetching in Mozilla: A Server-driven Approach," in *Web content caching and distribution*. Springer, 2004, pp. 283–291.

[47] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: Cross-platform User-interaction Record and Replay for the Fragmented Android Ecosystem," in *Proc. of ISPASS*, 2015.

[48] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction," in *Proc. of HPCA*, 2016.

[49] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption using Program Analysis," in *Proc. of ICSE*, 2013.

[50] S. He, Y. Liu, and H. Zhou, "Optimizing Smartphone Power Consumption through Dynamic Resolution Scaling," in *Proc. of MobiCom*, 2015.

[51] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," in *Proc. of MobiSys*, 2012.

[52] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing," in *Proc. of OOPSLA*, 2013.

[53] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators," in *Proc. of HPCA*, 2008.

[54] 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.

[55] T. Li, C. An, Z. Tian, A. T. Campbell, and X. Zhou, "Human Sensing Using Visible Light Communication," in *Proc. of MobiCom*, 2015.

[56] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining Energy-greedy API Usage Patterns in Android Apps: an Empirical Study," in *Proc. of MSR*, 2014.

[57] D. Lo, T. Song, and G. E. Suh, "Prediction-Guided Performance-Energy Trade-off for Interactive Applications," in *Proc. of MICRO*, 2015.

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

[59] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Domain Knowledge based Energy Management in Handhelds," in *Proc. of HPCA*, 2015.

[60] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP Chains on Handheld Platforms," in *Proc. of ISCA*, 2015.

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

[62] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative Energy Diagnosis for Mobile Devices," in *Proc. of Sensys*, 2013.

[63] 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.

[64] M. Pradel, P. Schuh, G. Necula, and K. Sen, "EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-guided Test Generation," in *Proc. of OOPSLA*, 2014.

[65] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *Proc. of PLDI*, 2011.

[66] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: Measuring Wireless Networks and Smartphone Users in the Field," in *SIGMETRICS Performance Evaluation Review*, 2011.

[67] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: A Language and Runtime System for Perpetual Systems," in *Proc. of SenSys*, 2007.

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

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

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

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

[72] ——, "The Role of the CPU in Energy-Efficient Mobile Web Browsing," in *Micro, IEEE*, 2015.

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

[74] ——, "WebCore: Architectural Support for Mobile Web Browsing," in *Proc. of ISCA*, 2014.