Event-Based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications

Yuhao Zhu Matthew Halpern Vijay Janapa Reddi

Department of Electrical and Computer Engineering
The University of Texas at Austin
yzhu@utexas.edu, matthalp@utexas.edu, vj@ece.utexas.edu

Abstract

*Mobile Web applications have become an integral part of our society. They pose a high demand for application quality of service (QoS). However, the energy-constrained nature of mobile devices makes optimizing for QoS difficult. Prior art on energy efficiency optimizations has only focused on the trade-off between raw performance and energy consumption, ignoring the application QoS characteristics. In this paper, we propose the concept of energy-efficient QoS (**eQoS**) to capture the trade-off between QoS and energy consumption. Given the fundamental event-driven nature of mobile Web applications, we further propose event-based scheduling as an optimization framework for eQoS. The event-based scheduling automatically reasons about users' QoS requirements, and accurately slacks the events' execution time to save energy without violating end users' experience. We demonstrate a working prototype using the Google Chromium and V8 framework on the Samsung Exynos 5410 SoC (used in the Galaxy S4 smartphone). Based on real hardware and software measurements, we achieve 41.2% energy saving with only 0.4% of QoS violations perceptible to end users.*

1. Introduction

Mobile applications are user-facing and highly interactive, unlike traditional background and batch-processing workloads. Guaranteeing satisfactory quality-of-service (QoS) experience for mobile end users becomes crucial. However, a single-minded pursuit of a performance-oriented mobile system design is infeasible, because today's mobile devices are energy constrained. As lithium-ion battery density starts plateauing [1], and battery form factor begins to mature [2], the total device energy budget (battery density $\times$ battery volume) is expected to saturate in the near future [84]. Energy efficiency is now the first-class design constraint.

To optimize for energy efficiency, some mobile systems have started adopting the heterogeneous multicore architecture that incorporates cores with different microarchitectures (e.g., out-of-order vs. in-order), each with various dynamic voltage and frequency scaling (DVFS) capabilities. An application is scheduled to the core and frequency setting that best trades off performance with energy consumption. Unfortunately, traditional scheduling techniques do not take into account the application QoS characteristics. Instead, they simply try to trade off raw machine performance (latency or throughput) for energy consumption. Such a QoS-agnostic approach creates two major energy efficiency problems:

- First, the system responsiveness may exceed the human acceptable limit–i.e., an end user's QoS expectation is violated. As a result, users abandon the service. For example, 40% of mobile users will abandon a webpage that takes more than 3 seconds to load [3]. All the energy consumed up until the service abandonment is then wasted.
- Second, the system may respond faster than end users' QoS expectations. Under this circumstance, a certain portion of the energy is wasted without any user-perceptible value. For example, human-computer interaction research shows that users can tolerate 100 ms latency for certain interactive tasks [60, 85]; however, we find that Android's `interactive` CPU governor [4] tends to finish interactive tasks faster than 100 ms by using the highest CPU performance, leading to unnecessary energy waste.

To address these QoS-related energy efficiency issues that are not well-captured by the traditional performance-energy trade-off, we propose a new concept called energy-efficient QoS (**eQoS**). eQoS requires appropriate scaling of the CPU's capability such that it provides "just enough" performance to meet end users' QoS expectations with minimal energy consumption. It emphasizes the QoS-energy trade-off as a new way of reasoning about the energy efficiency optimizations in the mobile Web domain.

In order to apply eQoS intelligently to trade off QoS for energy reduction, we require a step-by-step approach that imposes the following three requirements. First, it is important to systematically understand application QoS and how exactly QoS is affected by machine performance. We leverage an important observation that mobile Web applications are built atop the event-driven model, where various user interactions are translated to different application events. Our analyses show that two fundamental event-level properties, i.e., event intensity and event latency, reflect how user QoS experience is affected by machine performance. This further allows us to classify application events into three distinct QoS categories according to their intensity and latency characteristics.

Second, on the basis of understanding application QoS, it is important to design an eQoS-oriented runtime system that specifically leverages the QoS-energy trade-off. Simply ap-

plying traditional performance-energy optimizations may lead to suboptimal decisions. We propose a novel *event-based scheduling* mechanism that serves as the basis for trading-off QoS with energy consumption. The scheduler automatically reasons about users' QoS expectations based on the application's event characteristics, and accurately predicts the ideal core and frequency setting that guarantees satisfactory user QoS experience while achieving significant energy savings.

Third, it is also important to have a metric that can be used to quantitatively evaluate how good a particular eQoS optimization is compared to other eQoS optimizations. We propose such a metric, called QoS-per-Energy (QPE), which quantifies the trade-off between QoS and energy consumption. QPE can also be used to compare the eQoS optimization efficiency across different types of systems.

We demonstrate our work using Google Chromium (the Chrome browser's open source version) [5] on the Exynos 5410 SoC (used in Samsung Galaxy S4). Real system measurements show that an eQoS-oriented event-based scheduler saves 41.2% energy over always supplying the highest performance with only 0.4% more QoS violations. Compared to Android's `interactive` and `ondemand` governors, the event-based scheduler achieves 37.9% and 22.9% energy savings, respectively, with only 0.1% more QoS violations.

In summary, we make the following contributions:

- We propose eQoS. It is a framework for reasoning about the QoS-energy trade-off in interactive mobile applications.
- We propose event-based scheduling. It accounts for the fundamental event-driven nature of interactive mobile Web applications, and provides a framework to trade off QoS with energy consumption.
- We propose QPE. It is an eQoS metric that quantifies the trade-off between application QoS and energy consumption. QPE can be used in QoS-aware energy efficiency optimizations in the interactive mobile Web domain.

The paper is organized as follows. Sec. 2 presents the experimental methodology. Sec. 3 introduces the concept of eQoS, which motivates the need to orchestrate energy efficiency optimizations with application QoS in mobile systems. Sec. 4 systematically dissects application QoS from an event perspective, and identifies the key event characteristics that affect application QoS. Leveraging the event-level insights, Sec. 5 proposes an event-based scheduling framework that serves as the basis for trading off QoS with energy. Sec. 6 shows that our scheduler achieves better eQoS than traditional approaches. We present related work in Sec. 7, and conclude in Sec. 8.

2. Experimental Setup

In order to establish sound conclusions, we base our study on real measurements of contemporary hardware and software systems. This section describes our experimental setup, including mobile hardware/software platforms and data acquisition techniques for hardware- and software-based measurements.

Hardware Platform We use the ODroid XU+E development board [6] that hosts the Samsung Exynos 5410 SoC as the hardware platform. The Exynos 5410 SoC contains a representative big.LITTLE heterogeneous multicore CPU subsystem. The big.LITTLE system is divided into two homogeneous quad-core clusters, a big Cortex-A15 (A15) cluster and a little Cortex-A7 (A7) cluster, both manufactured using 28 nm technology. The four cores within a cluster can be individually enabled and disabled. The A15 processor is a 3-issue out-of-order core with 32 KB L1 instruction and data caches [7]. It operates from 800 MHz to 1.8 GHz at a 100 MHz granularity. The A7 processor is a dual-issue in-order core [8]. It has private L1 caches of the same size as in A15. It operates from 350 MHz to 600 MHz at steps of 50 MHz.

Software Infrastructure The ODroid board runs Android Version 4.2.2. We use Google's Chromium Web browser (Version 33.0) as the underlying runtime environment for the mobile Web applications studied in this paper. We modified the Timeline tool in the Chrome DevTools [9] as well as the V8 built-in profiler [10] to create an instrumentation tool that enables all the necessary timing instrumentations.

Energy Measurement The ODroid development board has built-in current sensing resistors for both the big and little cluster and the memory module. Each sense resistor is 10 mΩ. We use the National Instruments DAQ Unit X-series 6366 to simultaneously collect voltage measurements across the big and small CPU clusters, as well as the memory, at a rate of 1,000 samples per second. We did not observe any noticeable difference at 1 million samples per second. Thus, we preferred a lower sampling rate to enable faster measurement processing.

Reproducibility Interactive program behavior largely depends on both user input and its timing. It is important for ensuring reproducibility of inputs within interactive experiments. We embed all interactions into the benchmarked applications. This ensures that the same event sequences occur in each experiment trial. Each experiment has at least three trials. We observe less than 3% variation for almost all measurements.

3. eQoS: Trade-off Between QoS and Energy

Mobile system designs must optimize for energy efficiency. Conventional notions of energy efficiency typically trade off raw machine performance with energy consumption. For interactive applications, however, raw performance does not directly correspond to application QoS, and as such simply trading off performance with energy may lead to a suboptimal energy efficiency point. Therefore, to constructively reason about energy efficiency in the interactive application domain, we introduce a new concept called *energy-efficient QoS* (eQoS) that captures the QoS-energy trade-off, rather than the traditional performance-energy trade-off.

We illustrate the relationship between application QoS, performance, and energy savings in Fig. 1. Performance degrades from left to right on the *x*-axis. The left and right *y*-axes indicate QoS and energy savings, respectively. Foundational work

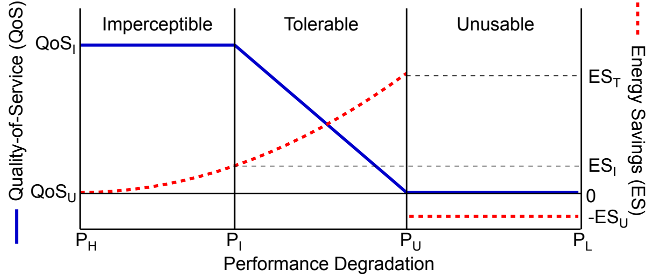


Fig. 1: The interplay between QoS, performance, and energy.

in human-computer interaction research [55, 60, 74, 76, 77, 85] indicates that interactive application QoS can be classified into three distinct states as machine performance degrades: *imperceptible* $[P_H, P_I]$, *tolerable* $(P_I, P_U]$, and *unusable* $(P_U, P_L]$.

In the imperceptible region, performance can degrade without any user-perceptible QoS loss while achieving more energy saving. Imperceptible QoS, QoS_I, is maintained until performance reaches P_I, the lowest performance capability that provides QoS_I. In the imperceptible region, supplying higher performance simply leads to more energy waste without adding any end-user value. For example, the most conservative approach to guarantee application QoS is to supply the peak performance of P_H; it leads to an energy waste of ES_I.

Beyond P_I, the application QoS enters the tolerable region, where the QoS is deteriorated as the performance reduces, but still remains tolerable. Any QoS could be acceptable in this region depending on the usage scenario or specific user pattern [92, 94]. Therefore, the tolerable QoS region exhibits a traditional performance-energy trade-off space.

As the performance further degrades, the QoS is eventually violated at P_U, where the user deems the QoS as unacceptable. P_U is the performance limit where users no longer feel engaged by the application. At P_U and beyond, users abandon the service. As a result, any energy consumed up until the service abandonment (ES_U) is wasted, because the underlying computation does not provide any utility to the user.

In summary, eQoS represents one of following three optimization objectives of a mobile system: 1) when the user QoS expectation is high, guaranteeing imperceptible QoS experience with the minimal energy by exploiting the performance slack between P_H and P_I, 2) when the user QoS expectation is low, guaranteeing usable QoS experience with the minimal energy by exploiting the performance slack between P_I and P_U, and 3) without specific user QoS expectation, allowing flexible performance-energy trade-off between P_I and P_U.

4. QoS in Mobile Web Applications

In this section, we introduce a general methodology to systematically identify the three QoS regions (imperceptible, tolerable, and unusable) and determine the P_I and P_U values in interactive mobile Web applications. By identifying and analyzing the applications using two fundamental event-level characteristics, i.e., event intensity and event latency, we show that applications fall into one of three categories depending on the two event characteristics (Sec. 4.1). We then leverage insights from human-computer interaction research, and use event intensity and latency characteristics to identify the P_I and P_U values for each application category (Sec. 4.2).

4.1. Application Event Characteristics

Web applications are built atop the event-driven execution model to achieve interactivity. Various user interactions, sensor inputs (e.g., gyroscope orientation), and application internal tasks (e.g., timers, completion of loading a webpage element) are translated to one or more applications events. Each event is registered with an event handler that is executed when the event is triggered. Internally, the browser employs a FIFO-like event queue where any newly generated events are placed at the end. A software thread continuously monitors the event queue and dequeues any available event from the head of the queue for processing, one event at a time. Therefore, events in the queue are processed in a synchronous manner.

The event-driven nature of mobile Web applications motivates us to analyze them from an event perspective. We identify two fundamental event-level characteristics: event intensity and event latency. *Event intensity* is the frequency of events triggered per second. *Event latency* is defined as the event execution time between when an event starts and when the results of the event becomes ready for the user to interact with. It indicates the responsiveness to an event. Let us explain how the two event characteristics affect the application QoS.

Event intensity reflect an application's prevailing form of human-computer interaction. With high event intensity, users are interacting with a high volume of repetitive events and having an "indulgent" session. Therefore, users interpret their QoS experience by the event throughput. With low event intensity, users interpret their QoS experience based on the application's responsiveness to each event, i.e., latency [60].

Typically, users have high tolerance for high-latency events because they are aware that a computationally intensive job is being processed [77]. As a result, their tolerance thresholds, by which they deem an application is imperceptible, tolerable, or unusable, are high. For low-latency tasks, such as typing or swiping, users expect a more seamless experience. As a result, their thresholds for QoS are lower [71].

To study event characteristics in detail, we examined a wide range of typical interactive mobile Web applications. Table 1 shows a selected subset of those applications. We considered several other applications, but do not include them due to space limitations. We considered extending prior work to meet our needs, but we found that the applications in prior work did not provide sufficient coverage of event intensity and latency. Thus, we needed to identify the applications in Table 1 that provided the coverage needed for discovering new insights from an application event perspective. We explain the distinction further in the related work section (Sec. 7).

Interactive mobile Web applications have strong pairings of

Table 1: Workload Description

Category	Event Intensity	Event Latency	(P_I, P_U)	Application Domain	Workload
Job Delay	Low (≪ 1/s)	High	(1, 10) s	PDF Rendering Photo Editing Cryptography Compression	Pdf.js CamanJS Crypto Zlib
Response Latency	Low (∼ 1/s)	Low	(50, 100) ms	MVC Apps	Backbone.js Ember.js GWT jQuery
				Web Browsing	google ebay sina
				Painting	Paper.js
FPS	High (∼ 30/s)	Low	(60, 30) fps	Web Gaming Animation	Doom Rain

event intensity and event latency. Their event characteristics predominantly fall into one of these three categories: low intensity and high latency, low intensity and low latency, or high intensity and low latency. Because of the synchronous event execution described earlier, high event intensity and high event latency conflict with each other, and as such we do not find applications that map to that specific combination.

Note that we focus on predominant application event characteristics. Applications may contain events of different categories. However, our analysis indicates that an application's event execution is primarily dominated by one main category.

We examine the applications' event characteristics using Fig. 2. We instrument the Web browser and gather the average event behavior of the applications running on the A15 processor at its highest frequency (1.8 GHz). Fig. 2a shows the average event intensity of the three event categories. There is strong distinction between categories. Fig. 2b shows the CDF of the event latency for each category. Each (x, y) point in the figure corresponds to the percentage of total events (x) that are at or below a particular execution latency (y). Once again, we see distinct behavior across the three categories. Fig. 2c illustrates how the applications are distributed in the intensity versus latency space, where event intensity increases along the x-axis from left to right and event latency increases along the y-axis from bottom to top. It confirms that applications map to three distinct categories.

4.2. Event Imperceptibility (P_I) and Usability (P_U) Values

We must identify P_I and P_U values for each application category. In this section, we explain how we determine them for the applications we examine. We introduce the applications, explain their use cases, and provide rationale for the P_I and P_U values we associate with each application category.

Although we describe our applications for the sake of completeness, the key takeaway from this section, however, is that once any application is mapped into one of the three categories, we can apply the imperceptibility (P_I) and usability (P_U) values that we have identified for other eQoS studies.

4.2.1. Low Event-Intensity, High Event-Latency

Applications with low event intensity and high event latency typically require minimal user interaction to initiate computationally intensive tasks, or jobs. These jobs are typical of "job delay" applications. Job delay applications have minimal user interaction. Also, they exhibit low event intensity, which on average is much less than 1 event per second (Fig. 2a). Due to the computational intensity involved with these applications, the event latency of this category of applications typically exceeds 5 seconds and can reach over 10 seconds (Fig. 2b).

We study four application domains that embody events with low intensity and high latency: PDF rendering, decryption/encryption, compression, and photo editing. Although these applications traditionally existed as browser plug-ins, they are increasingly ported to pure Web applications to provide a cleaner, faster, and arguably more secure Web experience [11]. For all four examples, we create a harness application with which the user interacts to trigger application functionalities.

Pdf.js is a JavaScript-based PDF renderer that is supported by Mozilla labs [12]. It has been adopted as the default PDF viewer by many browsers such as Firefox and Opera [13]. In our evaluation, we have the harness application render a 14-page paper [14], which is initiated by a button press activity.

CamanJS is an Instagram-like application that represents the emerging application domain of online photo editing fostered by content sharing and social network websites [15]. In our experiment, the application applies a complex filter (sharpening, color saturation, and blending) to an image.

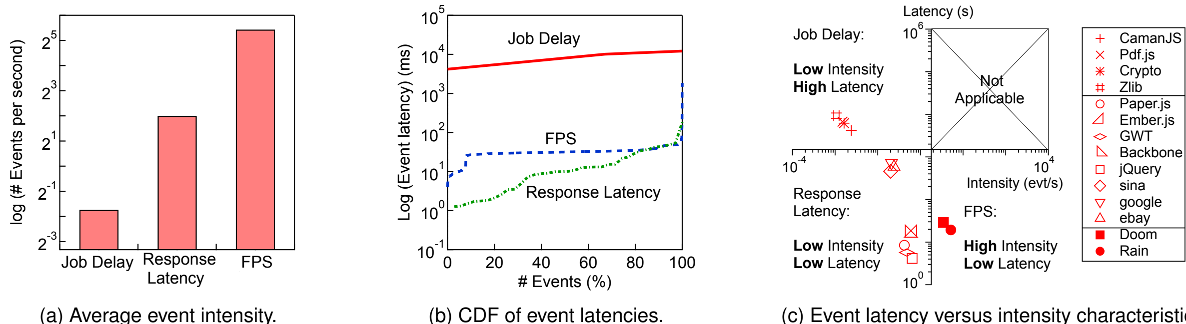
Zlib is a compression utility that reflects using data compression as a method to reduce network latency and maximize network bandwidth. For example, Google now enables data compression for Chrome in Android and iOS [16]. Our harness application encapsulates the widely used JavaScript implementation of the Zlib library [17]. It compresses 1000 KB of data, which is about the average size of today's webpages [18].

Crypto is a encryption application that reflects the trend that many mobile e-commerce and security-sensitive Web applications rely on data encryption and/or decryption to provide a secure user experience. We evaluate an application that encapsulates the widely used JavaScript-based RSA kernel from the JSBN library [19]. It encrypts and decrypts 960 KB data, which is close to an average webpage size [18].

P_I and P_U values User-experience studies [77] show that psychologically when users are aware of a computationally intensive interaction, they can subconsciously wait up to 1 second for the job to complete. Once the job execution exceeds 10 seconds, the user can no longer tolerate the delay. Therefore, we use 1 second and 10 seconds as the imperceptible QoS (P_I) and unacceptable QoS (P_U) boundaries, respectively, for the job delay applications with low intensity and high latency.

4.2.2. Low Event-Intensity, Low Event-Latency

Applications with low event intensity and low event latency



(a) Average event intensity. (b) CDF of event latencies. (c) Event latency versus intensity characteristics.

Fig. 2: Event characteristics for the different application categories.

usually originate from end-user interactions such as typing and touch gestures that are computationally lightweight. We call these applications "responsive latency applications." For these interactions, the users expect the responsiveness to be fluid. Human interactiveness indicates that the event intensity is typically low. Fig. 2a shows that the average event intensity for these applications is only about four events per second. Fig. 2b shows that the latency for about 60% of the events is lower than 10 ms, indicating low event latency.

We study three applications that have low event intensity and low event latency: painting, webpage browsing, and model-view-controller (MVC) based Todo list. These applications require frequent human interactions–similar to graphical user interface (GUI) environments, but in our case they belong to the mobile Web context.

Paper.js represents the popular online painting application domain. To date, there are over 500 painting applications in Google Play. We study Paper.js [20], a Web port of Adobe Illustrator's original scripting language Scriptographer [21]. We draw a series of curves using touch gestures for evaluation.

Web browsing represents a well-established mobile Web activity: surfing webpages. User QoS experience is strongly tied to the initial webpage load time in Web browsing. For instance, it is estimated that 79% of online shoppers will not return to the website with slow load time [22]. We benchmark webpages used by BBench [66] and Zhu and Janapa Reddi [96], and report the results for three webpages that represent diverse webpage behavior. They are www.ebay.com, www.google.com, and www.sina.com.cn. We delineate the webpage load time by the Web browser's internal onLoad event.

MVC is a popular design pattern for user interface applications [23]. Originally used in desktop applications, MVC is now widely used to create most mobile Web applications. We select an MVC-based todo list Web application from the TodoMVC project [24] because todo applications are common. The TodoMVC todo application demonstrates key MVC application features (e.g., data binding). Because MVC applications are typically developed using general application frameworks, we benchmark four different implementations of the todo list application, each using a different yet popular application framework. These frameworks include Backbone.js [25], Ember.js [26], Google Web Toolkit (GWT) [27], and jQuery [28]. We perform the following sequence of interactions on each of the applications: we first create a list item, mark it as completed, and empty the todo list altogether.

P_I and P_U values For events of a short period, prior research on user-experience [71] shows that users typically perceive visible changes within 50 ms. In the same study, users no longer perceive an instantaneous response after a 100 ms delay. Therefore, we use 50 ms and 100 ms as the imperceptible QoS (P_I) and unacceptable QoS (P_U) boundaries, respectively.

For Web browsing, however, we use 1 second and 3 seconds as P_I and P_U, respectively, specifically because of real-world usage scenarios. Google strongly promotes a 1 second webpage load time experience to not cause interruptions in users' flow of thought [29]. In addition, according to recent studies based on massive mobile Web users, 40% of Web users abandon webpages that do not load within 3 seconds [3].

4.2.3. High Event-Intensity, Low Event-Latency

The last category of applications has an order of magnitude higher event intensity than the previous two application categories (Fig. 2a). Such frequently executed events are usually triggered by an internal timer [30, 31] or requestAnimationFrame [32] callbacks. The high event intensity implies that user-experienced QoS is best evaluated by "event throughput."

The most prevalent example of event throughput is found in gaming or animation applications, where each timer event corresponds to the computation of a frame. Throughput is measured in the number of frames per second (FPS) [53]. Because of the high event intensity, this application category's event latency is low at only about 33 ms (Fig. 2b).

We studied two popular FPS applications. Doom is an example of Web gaming and Rain is an example of physically based animation. Traditionally, gaming and animations in the Web were created using Flash technologies; however, recent statistics show that over 90% of the graphics and animations in the Web are now developed using JavaScript and rendered to HTML5 canvas or SVG elements [33]. Doom and Rain both rely on these emerging Web technologies.

Doom is a popular first-person shooter game that has been ported to JavaScript [34]. It contains the Doom engine, which attempts to realistically render 3D graphics in a 2D plane. It is overall a computationally intensive task because it must perform binary space partitioning [63] and raycasting simul-

taneously. We benchmarked the game for 2 minutes using a prerecorded user action trace contained in the game.

Rain incorporates a complex and Web-based physics engine that simulates raindrops [35]. It is part of the Chrome Experiment [36]. We also benchmarked Rain for 2 minutes.

P_I and P_U values Prior research in gaming and animations shows that 60 and 30 FPS guarantee "seamless" and "just playable" user experience, respectively [56]. To evaluate eQoS, we used 60 FPS and 30 FPS as the imperceptible QoS (P_I) and usable QoS (P_U) boundaries, respectively.

4.3. Summary

Fig. 2c summarizes our applications' event characteristics. Most mobile Web application events fall into one of three distinct categories described in this section. Event intensities and latencies vary significantly from one category to another, which in turn dictates how the three QoS regions are mapped (P_I and P_U values). To optimize interactive event-driven Web applications, we must identify these events and leverage the event-level characteristics for eQoS optimization.

5. Event-Based Scheduling

We propose an event-based runtime scheduling mechanism to optimize for eQoS. Fig. 4 presents an overview of our event-based scheduling framework that is integrated into the Chromium Web browser. We first present our motivation for performing event-based scheduling at the event handler level (Sec. 5.1). We then provide a high-level design overview of the event-based scheduling framework (Sec. 5.2) and then describe its implementation details (Sec. 5.3). We target our scheduler at asymmetric architectures comprising both power-hungry out-of-order (big) cores and power-conserving in-order (little) cores, each capable of performing DVFS, because such systems are known to provide a large scheduling space [70].

5.1. Scheduling Unit

The scheduling unit in the event-based scheduler is the event handler. Whenever an event is triggered, a corresponding event handler is executed. Fig. 4 provides an example, showing how event handlers H1, H2, and H3 (in that order) are pushed into the event queue for execution. For events that share the same P_I and P_U values, we find that their event handlers have different execution latencies, and therefore lead to different performance slacks. We must treat each event handler differently and make scheduling decisions at that granularity.

We explain the variation in the event handlers' execution behavior using the Ember.js-based todo list application. Fig. 3 shows the sorted execution latencies of all the event handlers. The *x*-axis corresponds to the event handlers and the *y*-axis corresponds to the event handlers' execution latencies. The events fall into the low intensity and low latency category, and thus share the same P_I and P_U constraints. In this example, we assume that the performance target for the scheduler is to do "barely" better than the usability QoS threshold (P_U), which as described earlier is 100 ms for this event category (Sec. 4.2.2).

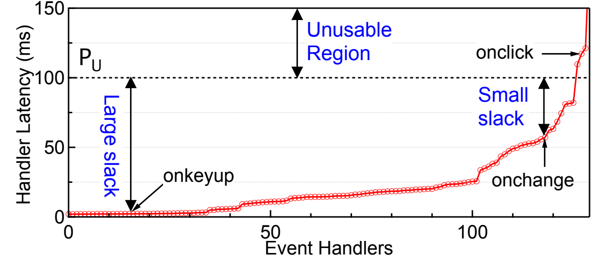


Fig. 3: Event handler variation in Ember.js todo list application.

We observe a large latency variation for the handlers in Fig. 3. We label three of the application's representative event handlers as the application executes: onkeyup, onchange, and onclick. The keyup event handler only processes one keystroke and therefore finishes execution very quickly in just 2 ms, which leaves a large amount of slack (98%) for the scheduler to exploit. In contrast, the onchange event handler adds one entry into the todo list. It requires about 50 ms for execution, which translates to only about 50% slack in performance. Lastly, the onclick event handler deletes all the entries in the todo list. The processing time exceeds P_U, and as such there is no opportunity to exploit performance slack. Instead, it requires a higher performance configuration, if available.

5.2. Scheduler Design Overview

The event-based scheduler predicts the ideal heterogeneous architecture execution configuration (i.e., a <*core*, *frequency*> tuple) whenever an event is triggered and the corresponding event handler is executed such that it "barely" meets the performance target with minimal energy consumption. The performance target could be expressed as achieving some level of QoS experience that is bound by P_I and P_U.

The scheduler consists of a simple dispatch front-end and scheduling back-end. The front-end *Dispatch* unit simply dequeues event handlers from the event queue one at a time in compliance with the synchronous, atomic event-driven execution model. It also extracts related event information from the application to pass to the back-end. The back-end consists of a *Detector*, *Model Constructor*, and *QoS Monitor*. The detector automatically identifies each event's QoS boundaries (i.e., P_I and P_U values), which impose constraints on how much performance slack the scheduler can exploit. The model constructor builds a performance and energy model for each event handler and predicts the latency and energy consumption for an event handler's execution. The models and the event QoS information are then fed into the monitor, which determines the architecture configuration for the event handler given the P_I and P_U constraints. We now describe the functionality of each, and how they interact as a whole scheduler.

Detector It identifies the P_I and P_U values for an event handler. It automatically detects the values based on event latency and event intensity information, but this is a two-step

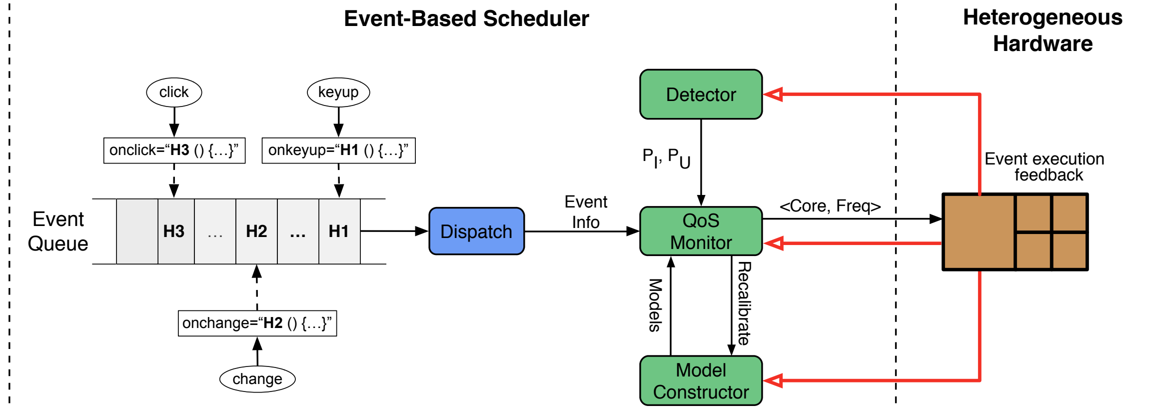


Fig. 4: Event-based runtime scheduling framework.

process. In step one, the scheduler profiles the event handler at the highest frequency on the big core to determine the event's category. Empirically, the detector deems the event handler as "high latency" if its latency is higher than 0.8 second, and vice versa. It deems an event handler as "high intensity" if its execution frequency is higher than 3 times per second, and vice versa. In step two, depending on the event intensity and latency values it observes, the detector determines the event handler's P_I and P_U constraints. Recall that Sec. 4.2 explains how we determine the P_I and P_U values for different event categories. Additionally, the detector persistently stores and recalls the runtime QoS profile information in an event profile file that is read whenever the Web browser is relaunched to enable cross-run optimization and amortize overhead effectively.

Model Constructor Due to the large variation across different event handlers shown in Fig. 3, the model constructor builds a performance and energy model for each event handler. For example, H1, H2, and H3 in Fig. 4 have their own models. We describe the model specifications in the next subsection.

QoS Monitor The monitor takes the predictive models along with the P_I and P_U values from the detector to determine the architecture configuration that should be used to execute a handler to satisfy an optimization objective. For example, if the goal is to minimize energy consumption in the imperceptible region, the scheduler selects the core and frequency combination that the model constructor predicts will consume the least energy while still delivering performance above P_I.

Furthermore, during application execution the monitor keeps monitoring event latencies and intensities on the hardware, and uses the information to adjust its prediction and scheduling decisions on the fly, similar to conventional feedback-driven optimizations [86]. We explain the detailed operation of the monitor in the next subsection. Intuitively, it is possible for the performance and energy models to underpredict or overpredict the architecture configuration. Under such circumstances, the monitor can decide to tune the predicted frequency or transition between big and little cores. If the models are deemed completely unusable, the monitor informs the model constructor to recalibrate the models.

5.3. Scheduler Implementation Details

Performance Model We construct performance models for big and little cores separately. Each model predicts the event handler execution latency under different frequencies. We use the classical DVFS analytical model initially proposed in [91], and employed in subsequent work, such as [90]:

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

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

Strictly speaking, $N_{dependent}$ is a function of f. However, precisely constructing a model that varies $N_{dependent}$ with f is complex and introduces a large calibration overhead at runtime. In our experiments, we find that it is feasible and necessary to trade model precision for performance. In particular, we find that treating $N_{dependent}$ as a constant is sufficient in our case.

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

Energy Model The energy model predicts the energy consumption of an event handler's execution. We construct the energy model on the basis of the performance model and the estimated power consumption. We derive the power estimation of all the core and frequency combinations by performing a profiling run and storing the results in a local power profile file

that is read by the Web browser upon every launch. Persistently storing and looking up the power profile file aligns with the Android standard [37]. Alternatively, we can dynamically derive the power consumption if power proxy counters, such as Running Application Power Limit (RAPL) [58], are available and exposed to software. In our case, a rough estimate of the power consumption is sufficient.

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

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

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

Overheads The QoS monitor accounts for scheduling overheads, which consist of two components: the overhead of the scheduling algorithm itself and the overhead of changing the architecture configuration (i.e, big/little core migration and/or frequency scaling). The scheduling algorithm's overhead is dominated by model construction, which only requires solving a two-variable linear system that imposes almost negligible overhead. For changing the architecture's configuration, we assume 100 μs for frequency scaling and 20 μs for switching cores, as indicated in [49, 50].

6. Evaluation

We first introduce our metric for evaluating eQoS optimizations (Sec. 6.1). We then describe the baseline schedulers that the event-based scheduler (EBS) compares against (Sec. 6.2), and validate the EBS's model accuracy (Sec. 6.3). We present a comprehensive evaluation of EBS using two important usage scenarios (Sec. 6.4). Our results cast two important implications. First, simply optimizing for EDP does not guarantee eQoS (Sec. 6.5). Second, having both big and little cores is strongly beneficial for eQoS optimizations (Sec. 6.6).

6.1. Metrics for Evaluating Event-Based Scheduling

We propose a new metric called *QoS per energy* (QPE) as a quantitative measure of eQoS to evaluate how well a system or a particular optimization technique balances application QoS with energy consumption. It is defined as follows:

$$QPE = \frac{QoS\ Score}{Energy\ Consumption}$$

"QoS score" is a measure of how performance affects QoS. It is a utility function that varies between 0 and 1. Ideally, one would want to use the least amount of energy to achieve the highest QoS score. Thus, a high QPE score indicates a better eQoS optimization than an optimization with a low QPE.

Imperceptible QoS score is defined as 1. Unusable QoS score is defined as 0. A QoS score in the tolerable region varies with performance. A recent mobile user study shows that the QoS in the tolerable region degrades linearly with performance [94]. Therefore, in our current definition, the tolerable QoS score also varies linearly between 1 and 0. The linearity may change depending on the actual QoS-performance relationship of specific usage scenarios. For instance, QoS is sometimes modeled as exponentially correlated with performance degradation in the network domain [61] and desktop applications [60]. Our formulation of QPE allows such customized QoS scores to be readily plugged in if needed.

Comparison to Other Metrics Energy consumption and QoS violations are two common metrics used to evaluate interactive systems. In our paper, we define a QoS violation as an event execution that exceeds a specified performance target, P_I or P_U. QoS violations are important to quantify in interactive mobile Web applications because they impact end user experience. In response latency and FPS applications (Table 1), a QoS violation is quantified by the percentage of events handlers whose QoS is violated. In job delay applications, event handlers have high latency, and thus it is more relevant to quantify the percentage that an event handler's latency exceeds the performance target.

Both of the above metrics, however, have limitations. Simply examining one metric without the other fails to comprehensively capture how a system trades off QoS with energy consumption. On one hand, a system with low performance would always achieve low energy consumption, but that can lead to unusable QoS. On the other hand, a system that consistently overprovisions performance will suffer from the fewest QoS violations, but that can cause excessive energy consumption due to exceeding the P_I boundary.

Properties The QPE metric has the following three properties that capture the fundamental QoS versus energy trade-off. First, QPE is monotonically decreasing beyond the imperceptible boundary, because the QoS score remains the same but the energy consumption keeps increasing. It reflects the fact that supplying higher performance beyond P_I simply leads to energy waste without adding any user-perceptible QoS value.

Second, QPE in the unusable region is 0 because the QoS value is 0. It reflects the fact that any performance worse than P_U is meaningless from a user QoS perspective due to likely service abandonment. Third, in the tolerable region where no hard QoS is imposed, QPE enables evaluating different performance-energy trade-offs. In the tolerable region, QPE is equivalent to the EDP metric because the QoS value varies linearly with the performance in our definition. Sec. 6.5 provides quantitative comparison between EDP and QPE.

6.2. Baseline Schedulers

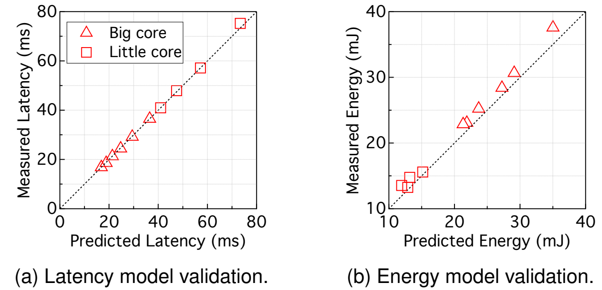
We compare EBS with the following four baseline schemes:

1. **Oracle-sched:** It is the oracle event-based scheduler that has *a priori* knowledge of all event handler latencies. It always maximizes the QPE score.
2. **Perf-sched:** It provides the highest performance (i.e., big core's highest frequency) to minimize QoS violation. It is the standard runtime policy for interactive applications.
3. **Energy-sched:** It achieves the lowest energy consumption during each event handler's execution.
4. **Interactive-sched:** It is the Android's `interactive` *cpufreq* governor designed specifically for interactive mobile applications [4]. In most Android smartphones, it is the default CPU governor. In our system, the `interactive` governor samples CPU utilization every 80 ms, and maximizes the CPU frequency if the CPU utilization is above 85%. Once triggered, it stays at the maximum frequency for at least 20 ms before re-evaluating the CPU utilization.
5. **Ondemand-sched:** It is the `ondemand` *cpufreq* governor [4] also commonly used in Android. It samples CPU utilization every 100 ms and maximizes CPU frequency if the CPU utilization is over 90% during the past sampling period. It gradually decreases the frequency by 100 MHz if the CPU utilization drops below 90%.

6.3. Prediction Model Accuracy

We validate our energy model against the real hardware measurement using the experiment setup described in Sec. 2. The energy model has an average error of 5.6%. Also, across all applications the performance model has an average error < 1%. Fig. 5 shows the model accuracy of Paper.js, which is representative of the median accuracy across all the applications. Both models are accurate enough to allow the event-based scheduler to distinguish between different architecture configurations. The accuracy justifies our simplification to trade off model accuracy for prediction speed (Sec. 5.3). Because of space limitations, we do not discuss the full data.

Performance model recalibration happens when the model mispredicts for more than four consecutive times. Across all benchmarks, the Ember.js- and GWT-based todo list applications incur the "most" model recalibrations—only twice over 600 event handler executions. Therefore, recalibration overhead in our event-based scheduler is negligible.



(a) Latency model validation. (b) Energy model validation.

Fig. 5: The latency and energy model accuracy for Paper.js, which has the median accuracy of all the applications.

6.4. Scheduling Scenarios

We evaluate various schedulers under two scheduling scenarios: scheduling for imperceptibility (P_I) and scheduling for usability (P_U). They characterize two important QoS experiences: "imperceptible delay" and "just usable". Let us now describe the two scheduling scenarios and explain the rationale of choosing them. Note that we use the P_I and P_U values listed in Table 1 as the imperceptibility and usability QoS thresholds.

1. **Scheduling for imperceptibility:** With abundant energy, or when the user expects high QoS, an eQoS-optimized system needs to perform a task "barely" better than the imperceptibility QoS threshold (P_I) using the least energy.
2. **Scheduling for usability:** Under a tight energy budget, or when the user QoS expectation is low, an eQoS-optimized system needs to perform "barely" better than the usability QoS threshold (P_U) using the least amount of energy.

Imperceptibility We first compare the QoS violation and energy consumption of EBS against other runtime schemes to understand EBS' behavior. We then use QPE to summarize EBS' benefit in achieving eQoS. Fig. 7a, Fig. 7b, and Fig. 7c show the QoS violation, energy consumption, and QPE score under different scheduling schemes, respectively. The QPE scores are normalized to Oracle-sched. The EBS_Imp bars correspond to the EBS when scheduling for imperceptibility.

When scheduling for imperceptibility, the event-based scheduler provides nearly equivalent QoS satisfaction as Perf-sched, which always provides the best performance to minimize QoS violations. Fig. 7a shows that EBS consumes only 0.4% more QoS violation than Perf-sched. Meanwhile, Fig. 7b

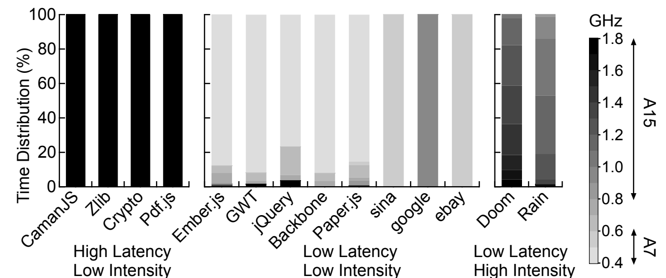
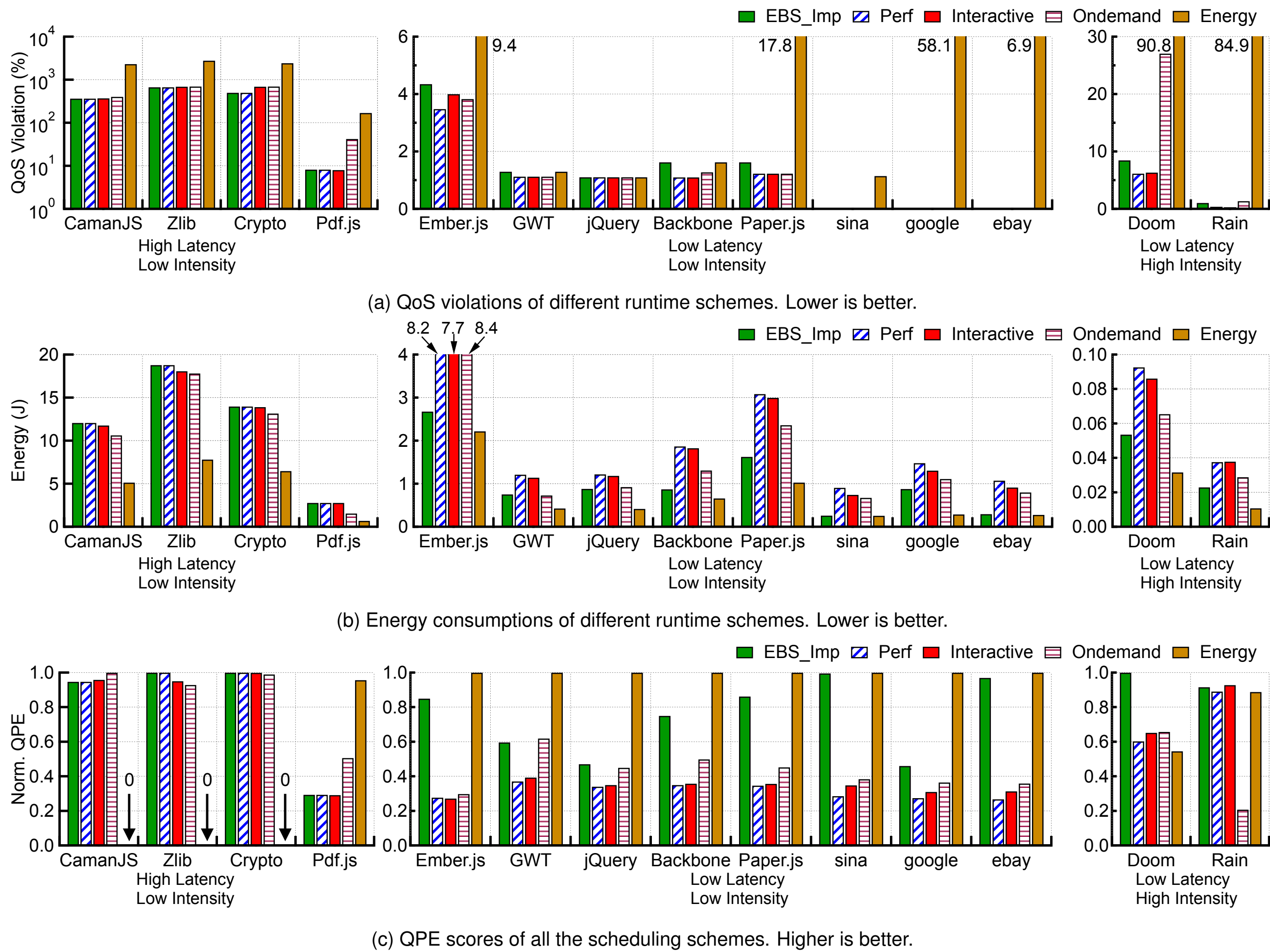


Fig. 6: The architecture configuration distribution of EBS scheduling for imperceptibility. Darker colors indicate higher performance.



(a) QoS violations of different runtime schemes. Lower is better.

(b) Energy consumptions of different runtime schemes. Lower is better.

(c) QPE scores of all the scheduling schemes. Higher is better.

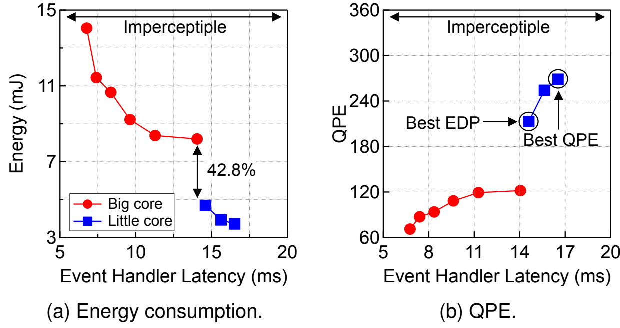
Fig. 7: QoS violations, energy consumptions, and QPE scores of different scheduling schemes when scheduling for impercepbility.

shows that EBS achieves on average 41.2%, up to 72.4%, energy savings. To understand the sources of energy savings, Fig. 6 illustrates the architecture configuration distribution of EBS. Except high-latency, low-intensity applications (first group), EBS extensively leverages the lower performance configurations. This means that EBS can deliver near-optimal QoS without always providing the highest performance, and thus maintain a low-energy footprint. Accordingly, EBS significantly improves the QPE over Perf-sched, as Fig. 7c shows.

EBS also significantly outperforms both of the OS governors. On average, EBS achieves 22.9% and 37.9% energy savings over Ondemand-sched and Interactive-sched, respectively, with about 0.1% more QoS violations. This is because the OS schedulers tend to overprovision the CPU performance, which is not needed by many applications. The most representative example is the low-latency, low-intensity applications (second group). They incur about 80% CPU utilization. In respond to the high CPU utilization, both Interactive-sched and Ondemand-sched schedule the processor under the big core with high frequency, effectively delivering similar performance as Perf-sched. However, Fig. 6 shows that EBS can identify the performance slack, and it schedules most of the event handlers using the little core while still meeting P_I. As a result, Fig. 7c shows that EBS achieves a much higher QPE score than Interactive-sched and Ondemand-sched.

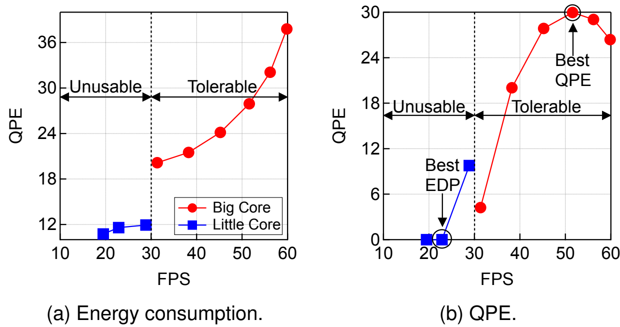
Compared to Energy-sched, which minimizes energy consumption without QoS guarantees, EBS reduces the QoS violations by 72.0%. For three of the high-latency, low-intensity (first group) applications (CamanJS, Zlib, and Crypto), Energy-sched even violates the usability threshold P_U. Thus, although Energy-sched consumes less energy for these applications, their QPEs are zero as Fig. 7c shows, indicating that the underlying scheduler provides no user value. For other applications, EBS either outperforms, or comes close to, Energy-sched.

Usability When scheduling for usability, i.e., meeting the P_U target using minimal energy, EBS achieves 55.4%, 52.9%, and 41.4% energy savings over Perf-sched, Interactive-sched, and Ondemend-sched, respectively, with nearly equivalent QoS violations ($< 0.1\%$). The energy savings are larger than scheduling for imperceptibility (41.2%, 37.9%, and 22.9% savings for the three schedulers, respectively). This is because scheduling for P_U requires lower performance than scheduling for P_I, and leaves more performance slack. Perf-sched, Interactive-sched, and Ondemand-sched are agnostic to the



(a) Energy consumption. (b) QPE.

Fig. 8: Energy consumption and QPE of the Ember.js todo app.



(a) Energy consumption. (b) QPE.

Fig. 9: Energy consumption and QPE of Rain.

QoS target. They can not effectively leverage the larger performance slack, and incur more performance overprovision. EBS, in contrast, can adapt to the changing performance requirement by leveraging lower performance configurations more frequently, and it consumes less energy. As compared to Energy-sched, EBS reduces the QoS violation by about 50%.

6.5. EDP Optimization versus eQoS Scheduling

Although QPE is equivalent to EDP in the tolerable region due to the linearity between QoS and performance, scheduling for eQoS is different from simply optimizing for EDP. Intuitively, this is because the best EDP configuration might be in the unusable or imperceptible region. Therefore, an EDP-oriented scheduler may lead to QoS violations or consume unnecessarily high energy, defeating the purpose of eQoS optimization.

We further explain the distinction between EDP and eQoS using the Ember.js-based todo list application and Rain. For each application, we select a representative event that has the median latency of all the event executions. Fig. 8b and Fig. 9b present the QPE scores under different architecture configurations for the selected event of the two applications, respectively. The x-axis corresponds to the event handler latencies (or FPS for Rain), and the y-axis corresponds to the QPE values.

For Ember.js, the best EDP configuration and best QPE configuration (i.e., the best eQoS-optimized configuration) do not match. This is because all the configurations achieve the imperceptible QoS experience. Because the QoS score in the imperceptible region is 1, the best QPE is achieved when the energy, instead of EDP, is minimized. Scheduling for the best EDP would select a higher performance configuration and consumes more energy without improving user-perceptible QoS. For Rain, the best EDP and QPE configuration do not match either. This is because the best EDP configuration does not provide enough performance to make the application usable. Therefore, the QoS score and the QPE result are zero.

6.6. Big/Little Architecture Effect

Our results indicate that a big/little heterogeneous architecture is beneficial for eQoS optimizations. Having only a big (or little) core reduces the optimization opportunity. For low-latency, low-intensity applications (second group), they benefit from having a little core. This is because a little core can already achieve an imperceptible QoS experience for most of the event executions while saving significant energy compared to a big core. For example, Fig. 8a shows the energy consumption and event latency of Ember.js under various architecture configurations. Even the weakest little core performance meets the imperceptible QoS target. Using the little core in this case saves at least 42.8% energy over the big core. Fig. 7c further confirms that EBS spends over 90% of the execution time on the little core for the second group applications.

In contrast, applications in the first and third group benefit from having a big core. This is because a little core fails to deliver an imperceptible QoS experience. For example, Fig. 9a shows the energy consumption and FPS of Rain under various architecture configurations. All of the little core configurations fail to achieve a tolerable QoS (30 FPS), eventually wasting energy upon service abandonment. Fig. 6 further shows that EBS leverages various frequency settings from the big core for first- and third-group applications.

7. Related Work

(e)QoS in Computer Architecture QoS has long been an important topic in computer architecture research. It has been applied to multicore memory schedulers and datacenters, in which QoS is typically manifested as fairness [75] and service-level agreements (SLA) [59], respectively. Our work focuses on QoS in the interactive mobile domain [56, 71], in which our contribution is to characterize the relationship between QoS and energy efficiency, and propose the term eQoS that captures the trade-off between QoS and energy consumption.

QoS-Oriented Energy Optimization Techniques for trading QoS for energy efficiency fall into two major categories: approximation and performance relaxation. Approximation techniques trade off precision for energy consumption (performance), such as applying inexact image filters [83].

Our event-based scheduling utilizes performance relaxation techniques that do not sacrifice precision, but exploit the gap between human perception and processor performance. It advances the existing performance slacking techniques in the interactive/mobile space in two key ways. First, traditional performance slacking techniques partition the QoS spectrum into either totally imperceptible or totally unusable. Such simplification does not consider the tolerable QoS in realistic

user experience [77, 85]. Our event-based scheduling allows exploiting the performance-energy trade-off in the tolerable region, more accurately reflecting the real user experience, and increases the energy optimization opportunities.

Second, the event-based scheduling framework does not rely on each application specifying the QoS requirements; rather, it automatically detects them based on the fundamental application event-level characteristics. As such, the scheduling framework is generally applicable to interactive mobile Web applications, rather than prior art that individually focuses on one application domain at a time, such as FPS applications [62, 93], and responsive latency applications [52, 60, 95].

Timer coalescing [51] used in OS X Mavericks also exploits the performance slack for energy savings, similar to our event-based scheduling. It postpones noncritical timers and coalesces them for batch executions to increase the processor idle time for energy savings. However, timer coalescing applies only to timers in Apple's native applications (or OS processes), whereas our EBS framework is not limited to timer events, but can apply to any event-driven applications.

Single-ISA/DVFS Scheduling The event-based scheduling differs from existing single-ISA scheduling and DVFS techniques in two key aspects: scheduling unit and scheduling objective. First, the scheduling unit in existing techniques is either interval based (fixed-instruction interval [70, 72, 73, 78, 81] or fixed-time interval [57, 65, 79, 88, 89]) or a code segment (e.g., critical sections, lagging threads, application kernels [54, 68, 69, 87]). The scheduling unit in the event-based scheduling is the event handler in interactive mobile Web applications. Event handlers correspond to user interactions and let us directly optimize for user QoS experience.

The scheduling objectives in existing techniques are typically architecture-level energy-efficiency metrics such as energy, EDP [64], and million-instructions-per-joule (MIPJ) [89]. These metrics trade off raw performance instead of QoS with energy. Therefore, they may lead to executions that fall into the imperceptible or unusable QoS regions and waste energy. On the contrary, the event-based scheduler explicitly considers the P_I and P_U constraint in order to guarantee satisfactory QoS experience. In addition, we also propose a metric called QPE, which could be directly used as a scheduling objective for trading off QoS with energy consumption.

Mobile Workload Suites and Characterization Our study on interactive applications is driven by understanding application QoS requirements from an application events perspective, which is not the focus in the majority of existing work [38–45, 67, 80, 82]. Other benchmarks consider only a particular form of QoS. For example, BBench [66] considers the webpage load time as the QoS constraint for Web browsing; the Web latency benchmark [46] considers the event latency of user actions to webpage elements; the GFXBench [47] and BaseMark [48] benchmarks consider FPS. However, our workload characterization efforts lead to a general methodology to identify QoS constraints of a wide range of applications.

8. Conclusion

Mobile system designs today must meet two conflicting goals: achieving energy efficiency and delivering satisfactory user QoS experience. In this paper, we propose eQoS, which serves as a general framework for reasoning about the energy-efficiency trade-off in interactive mobile Web applications. We show that by understanding the trade-off between user QoS experience, performance, and energy consumption, and leveraging the inherent event-driven execution model of interactive mobile Web applications, we can achieve satisfactory end-user QoS experience while avoiding always provisioning the highest performance through a novel event-based scheduler. We demonstrate a working prototype using the Google Chromium and V8 framework on a Samsung Exynos 5410 SoC. Real hardware and software measurements show that the event-based scheduling optimizing for eQoS achieves 41.2% energy saving with only 0.4% of perceptible QoS violations.

Acknowledgments

We are thankful to our colleagues in industry and academia, as well as anonymous reviewers for the many comments that have contributed to this work. This work is supported by Intel Corporation, AMD Corporation, Samsung, and Google. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Battery Statistics. http://batteryuniversity.com/learn/article/battery_statistics
[2] Size matters for connected devices. http://goo.gl/Mcocet
[3] Kissmetrics: "How loading time affects your bottom line". https://blog.kissmetrics.com/loading-time/
[4] Android CPUFreq Governors. http://goo.gl/Mlr9U1
[5] Chromium browser. http://www.chromium.org/Home
[6] Ordroid XU+E Development Board. http://hardkernel.com/main/products/prdt_info.php?g_code=G137463363079
[7] Exploring the Design of the Cortex-A15 Processor. http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf
[8] Enabling Mobile Innovation with the Cortex-A7 Processor. http://www.arm.com/files/downloads/Enabling_Embedded_Innovation_with_the_Cortex-A7_Processor.pdf
[9] Performance Profiling with the Timeline. https://developers.google.com/chrome-developer-tools/docs/timeline#frames_mode
[10] V8 Profiler. http://code.google.com/p/v8/wiki/V8Profiler
[11] Browsing Without Plug-ins. http://blogs.msdn.com/b/ie/archive/2011/08/31/browsing-without-plug-ins.aspx
[12] Pdf.js. http://mozilla.github.io/pdf.js/
[13] How Fast is PDF.js. http://goo.gl/b1bfVL
[14] Pdf.js demo. http://mozilla.github.io/pdf.js/web/viewer.html
[15] CamanJS. http://camanjs.com/
[16] Google Enables Data Compression for Chrome in Android and iOS. http://goo.gl/qR03iV
[17] Zlib Data Compression Library. https://github.com/kripken/emscripten/tree/master/tests/zlib
[18] The Average Web Page is Now 1 MB. http://goo.gl/Q0xLxl
[19] RSA and ECC in JavaScript. http://www-cs-students.stanford.edu/~tjw/jsbn/
[20] Paper.js. http://paperjs.org/
[21] Scriptographer. http://scriptographer.org/
[22] Speed is a Killer. http://blog.kissmetrics.com/speed-is-a-killer
[23] Steve Burbeck, Applications programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC). http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

[24] TodoMVC. http://todomvc.com/
[25] Backbone.js Todo List. http://todomvc.com/architecture-examples/backbone/
[26] Emberjs.js Todo List. http://todomvc.com/architecture-examples/emberjs/
[27] Gwt Todo List. http://todomvc.com/architecture-examples/gwt/
[28] Jquery Todo List. http://todomvc.com/architecture-examples/jquery/
[29] Delivering the sub one second rendering experience. http://developers.google.com/speed/docs/insights/mobile
[30] Timers specification. http://goo.gl/hU9pI1
[31] HTML Event Types. http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-eventgroupings-htmlevents
[32] W3C: Timing Control for Script-based Animations. http://www.w3.org/TR/animation-timing/
[33] Web Graphics Trends in 2013. http://www.html5canvastutorials.com/articles/web-graphics-trends-in-2013/
[34] Doom. http://kripken.github.io/boon/boon.html
[35] Rain Simulation. sheepeuh.com/rain
[36] Chrome Experiment. http://www.chromeexperiments.com/
[37] Power Profiles for Android. https://source.android.com/devices/tech/power.html#
[38] Browsing Bench. http://www.eembc.org/benchmark/browsing_sl.php
[39] BrowserMark Benchmark. http://browsermark.rightware.com/
[40] Vellamo Benchmark. http://www.quicinc.com/vellamo/
[41] AnTuTu Benchmark. http://www.antutulabs.com/downloads.html
[42] SunSpider JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html
[43] Octane Benchmark. https://developers.google.com/octane/benchmark
[44] Kraken Benchmark 1.1. http://krakenbenchmark.mozilla.org/
[45] GeekBench 3.0 Benchmark. http://www.primatelabs.com/geekbench/
[46] Web Latency Benchmark. http://google.github.io/latency-benchmark/
[47] GFXBench Benchmark. https://gfxbench.com/result.jsp
[48] BaseMark X. http://www.rightware.com/consumer/basemark-x/
[49] "System Software for ARM big.LITTLE Systems," in *ARM Whiltepaper*, 2011.
[50] "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," in *ARM Whiltepaper*, 2013.
[51] Apple, power Efficiency in OS X. https://www.apple.com/media/us/osx/2013/docs/OSX_Power_Efficiency_Technology_Overview.pdf
[52] M. Bi, I. Crk, and C. Gniady, "IADVS: On-demand Performance for Interactive Applications," in *Proc. of HPCA*, 2010.
[53] K. Brownlow, "Silent Films: What was the Right Speed?" *Sight & Sound*, 1980.
[54] 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.
[55] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The Information Visualizer: An Information Workspace," in *Proc. of CHI*, 1991.
[56] M. Claypool, K. Claypool, and F. Damaa, "The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games," in *Multimedia Computing and Networking*, 2006.
[57] 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.
[58] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le, "RAPL: Memory Power Estimation and Capping," in *Proc. of ISLPED*, 2010.
[59] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proc. of SOSP*, 2007.
[60] Y. Endo, Z. Wang, J. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," in *Proc. of OSDI*, 1996.
[61] M. Fiedler, T. Hossfeld, and P. Tran-Gia, "A Generic Quantitative Relationship between Quality of Experience and Quality of Service," in *IEEE Network*, 2010.
[62] K. Flautner and T. Mudge, "Vertigo: Automatic Performance-Setting for Linux," in *Proc. of OSDI*, 2002.
[63] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On Visible Surface Generation by a priori Tree Structures," in *Proc. of SIGGRAPH*, 1980.
[64] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," in *IEEE Journal of Solid-State Circuits*, 1996.
[65] D. Grunwald, P. Levis, K. Farkas, C. M. III, and M. Neufeld, "Policies for Dynamic Clock Scheduling," in *Proc. of OSDI*, 2000.
[66] 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.
[67] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A Mobile Benchmark Suite for Architectural Simulators," 2014.
[68] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *Proc. of ASPLOS*, 2012.
[69] ——, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *Proc. of ISCA*, 2013.
[70] 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.
[71] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown, "Attention web designers: You have 50 milliseconds to make a good first impression!" in *Behaviour & information technology*, 2006.
[72] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite Cores: Pushing Heterogeneity into a Core," in *Proc. of MICRO*, 2012.
[73] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting Performance Impact of DVFS for Realistic Memory Systems," in *Proc. of MICRO*, 2012.
[74] R. B. Miller, "Response Time in Man-Computer Conversational Transactions," in *Proc. of AFIPS Fall Joint Computer Conference*, 1968.
[75] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proc. of MICRO*, 2007.
[76] B. A. Myers, "The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces," in *Proc. of CHI*, 1985.
[77] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.
[78] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Trace Based Phase Prediction For Tightly-Coupled Heterogeneous Cores," in *Proc. of MICRO*, 2013.
[79] V. Pallipadi and A. Starikovskiy, "The Ondemand Governor: Past, Present, and Future," in *Proc. of Linux Symposium*, 2006.
[80] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, Energy Characterizations and Architectural Implications of an Emerging Mobile Platform Benchmark Suite-MobileBench," in *Proc. of IISWC*, 2013.
[81] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread Motion: Fine-Grained Power Management for Multi-Core Systems," in *Proc. of ISCA*, 2009.
[82] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated Construction of JavaScript Benchmarks," in *Proc. of OOPSLA*, 2011.
[83] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-Tuning Approximation for Graphics Engines," in *Proc. of MICRO*, 2013.
[84] F. Schlachter, "No Moore's Law for Batteries," in *Proc. of National Academy of Science of the United States of America*, 2013.
[85] B. Shneiderman, *Designing the User Interface*. Addison-Wesley, 1992.
[86] M. D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization (Keynote Talk)," in *Proc. of DYNAMO*, 2000.
[87] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," in *Proc. of ASPLOS*, 2009.
[88] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo, "User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices," in *Proc. of DAC*, 2014.
[89] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU energy," in *Proc. of OSDI*, 1994.
[90] 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.
[91] F. Xie, M. Martonosi, and S. Malik, "Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits," in *Proc. of PLDI*, 2003.
[92] Y. Xu, M. Lin, H. Lu, G. Cardone, N. D. Lane, Z. Chen, A. T. Campbell, and T. Choudhury, "Preference, Context and Communities: A Multi-faceted Approach to Predicting Smartphone App Usage Patterns," in *Proc. of ISWC*, 2013.
[93] L. Yang, R. P. Dick, G. Memik, and P. Dinda, "HAPPE: Human and Application-Driven Frequency Scaling for Processor Power Efficiency," *Mobile Computing, IEEE Transactions on*, 2013.
[94] Z. Zhao, M. Zhou, and X. Shen, "SatScore: Uncovering and Avoiding a Principled Pitfall in Responsiveness Measurements of App Launches," in *Proc. of UbiComp*, 2014.
[95] Y. Zhu and V. J. Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems," in *Proc. of HPCA*, 2013.
[96] ——, "WebCore: Architectural Support for Mobile Web Browsing," in *Proc. of ISCA*, 2014.