PES: Proactive Event Scheduling for Responsive and Energy-Efficient Mobile Web Computing

Yu Feng
yfeng28@ur.rochester.edu

Yuhao Zhu
yzhu@rochester.edu

Department of Computer Science, University of Rochester
https://www.cs.rochester.edu/horizon

Abstract

Web applications are gradually shifting toward resource-constrained mobile devices. As a result, the Web runtime system must simultaneously address two challenges: responsiveness and energy-efficiency. Conventional Web runtime systems fall short due to their *reactive* nature: they react to a user event only after it is triggered. The reactive strategy leads to local optimizations that schedule event executions one at a time, missing global optimization opportunities.

This paper proposes Proactive Event Scheduling (PES). The key idea of PES is to proactively anticipate future events and thereby globally coordinate scheduling decisions across events. Specifically, PES predicts events that are likely to happen in the near future using a combination of statistical inference and application code analysis. PES then speculatively executes future events ahead of time in a way that satisfies the QoS constraints of all the events while minimizing the global energy consumption. Fundamentally, PES unlocks more optimization opportunities by enlarging the scheduling window, which enables coordination across both outstanding events and predicted events. Hardware measurements show that PES reduces the QoS violation and energy consumption by 61.2% and 26.5%, respectively, over the Android's default `Interactive` CPU governor. It also reduces the QoS violation and energy consumption by 63.1% and 17.9%, respectively, compared to EBS, a state-of-the-art reactive scheduler.

ACM Reference Format:
Yu Feng and Yuhao Zhu. 2019. PES: Proactive Event Scheduling for Responsive and Energy-Efficient Mobile Web Computing. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322248

1 Introduction

The landscape of mobile computing has experienced a tremendous transformation over the past decade. A 2018 study shows that mobile devices have surpassed traditional devices and become the most pervasive personal computing platform [25]. The key enabler behind this transformation is the advancement in Web technologies, which provide a platform-independent way for mobile users to interact with the Internet while greatly improving developers' productivity. It is estimated that over two-thirds of the US mobile traffics are contributed by Web applications [60].

Two significant but conflicting challenges stand in the way of the future mobile Web: responsiveness and energy-efficiency. Responsiveness of mobile Web applications impacts user quality-of-service (QoS), and has significant financial implications. Amazon estimates that a one-second delay in webpage load time could translate to $1.6 billion lost in sales annually because mobile users abandon a Web service altogether if the webpage is deemed unresponsive [11]. However, a single-minded pursuit of performance to improve responsiveness is unscalable due to the tight energy budget of mobile devices, which are inherently constrained by the battery capacity without an external power supply [5].

To reconcile responsiveness with energy-efficiency, numerous prior work [30, 43, 55, 69, 71, 73] has exploited the heterogeneous Asymmetric Chip-Multiprocessor (ACMP) architecture that has been widely adopted by today's mobile hardware vendors such as Samsung [10], Qualcomm [16], and Apple [62]. The heterogeneities of ACMP, including different core types and frequency settings, expose a large performance-energy trade-off space. Although different in design and implementation, today's ACMP schedulers share one common idea: consume "just enough" energy for a given responsiveness target (deadline). In particular, since mobile applications are event-driven where state transitions are triggered only by events such as user interactions, the scheduling decisions are applied at an event-granularity.

However, a fundamental limitation of existing approaches is that they are *reactive* by nature in that they provision hardware resources to an event only after it has been triggered. Coupled with the reactive strategy is their *localized optimizations* that schedule events one at a time without accounting for the dynamics of future events. Collectively, existing schedulers miss great optimization opportunities due to the limited event scheduling scope and the inability to coordinate scheduling decisions across events.

Our key idea is that Web runtime systems can significantly improve application responsiveness and energy-efficiency by *proactively anticipating future events* and thereby *globally coordinating scheduling decisions across events*. To that end, we propose Proactive Event Scheduling (PES), which continuously predicts events that are likely to happen in the near future and coordinates event executions across both outstanding and predicted events. In particular, PES speculatively executes future events in a way that satisfies the deadlines of all the events while minimizing the global energy

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322248

consumption. Fundamentally, PES enlarges the "event scheduling window" and unlocks more optimization opportunities, similar to how microarchitecture speculative techniques increase the instruction window and offer more scheduling opportunities [37, 40].

Critical to the event prediction scheme in PES is the combination of statistical inference and program analysis. Through characterizing real-world user interactions, we find that user events within an interaction session exhibit strong temporal correlation, which allows us to infer future events from past events. The accuracy of such a prediction strategy can be further improved with program analysis. The intuition is that program control flow analysis helps narrow down all possible next events mandated by the application logic, tightening the prediction space used by the statistical inference model. We propose a hybrid learning-analytical approach that accurately predicts user event sequences with low overhead.

Leveraging the predicted event sequences, we introduce a new scheduling algorithm that coordinates pending events with predicted events for energy and QoS optimizations. The scheduler wisely schedules events to different ACMP configurations to minimize the global energy consumption while satisfying individual events' QoS requirements. We find that this scheduling task can be formulated as an integer linear programming (ILP) problem, which can be efficiently computed on the fly with near-optimal solutions.

We integrate PES with Google's open-source Chromium Web browser engine [7]. We evaluate PES using the ODROID XU+E [15] development board, which contains the Exynos5410 SoC that is used in Samsung Galaxy S4 smartphone. Based on real hardware measurements, PES achieves 26.5% energy savings and 61.2% QoS improvements over the Android's default `Interactive` CPU governor. PES also achieves 17.9% energy savings and 63.1% QoS improvements over EBS [69], a state-of-the-art reactive scheduler.

In summary, this paper makes the following contributions:

- We quantitatively demonstrate the inefficiencies of existing reactive schedulers for mobile applications.
- We propose to combine statistical inference with application analysis to predict future events. The predictor achieves high prediction accuracy while naturally adapting to different user behaviors and application contents.
- We introduce a new scheduling framework, PES, that simultaneously improves responsiveness and energy-efficiency of the mobile Web applications. PES proactively speculates user events and globally coordinates event executions based on constrained optimizations.
- Our evaluation results show that PES achieves significant energy savings while reducing QoS violations compared to existing schedulers, and is closed to oracle.

The rest of the paper is organized as follows. Sec. 2 describes the scope of mobile Web and introduces the background of Web runtime. Sec. 3 presents the experimental methodology. Sec. 4 characterizes real mobile Web applications to motivate the need for a proactive scheduler. Sec. 5 describe the design and implementation of PES. Sec. 6 quantifies the benefits of PES against other existing scheduling mechanisms. Sec. 7 puts PES in the broad content of mobile (Web) optimizations. Sec. 8 discusses the limitation and future developments of PES, and Sec. 9 concludes the paper.

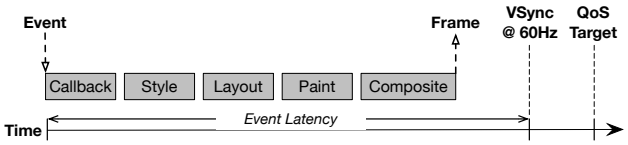


Fig. 1: The user QoS is dictated by the event latency, which, if exceeds the QoS target (deadline), degrades responsiveness.

2 Background

We first discuss the broad scope of Web computing and introduce its fundamental event-driven execution model. We also describe how QoS is evaluated in the mobile Web.

Web Applications Broadly, Web applications are applications that are developed using core Web languages including HTML, CSS, and JavaScript. Web applications not only include over 1.6 billion webpages [4] that are accessed through Web browsers, but also "hybrid" applications that are internally rendered by a Web browser engine and are wrapped by a native shell [12, 17]. The combination of conventional webpages and hybrid applications accounts for a majority of today's mobile applications [59], and contributes to more than two-thirds of today's mobile Internet traffic [60].

Web applications are platform-independent in that they rely on the Web browser engine as a virtual machine or a runtime system that dynamically translates HTML, CSS, and JavaScript to re-target different mobile platforms. Although beneficial to development productivity, the dynamic translation layer introduced by the Web runtime incurs significant compute overhead. Prior work has shown that mobile Web applications demands over 80% CPU usage [38], which in turn dominates the energy consumption of the mobile device [34]. Reducing the compute energy consumption while improving user-experience is thus the main goal of our work.

Event-Driven Execution Model Mobile Web adopts the event-driven execution model where user interactions (e.g., tapping) are translated to application events (e.g., `touchstart`) defined by the Document Object Model (DOM) and implemented by JavaScript. Each event is registered with an event handler (i.e., callback function) that is executed when the associated event is triggered.

The result of an event's callback execution is then passed to the browser's rendering engine, in which each event goes through a sequence of processing stages, such as style resolution, layout, paint, and composite [42], to produce a frame as a result of the user interaction. In the end, the browser submits the frame to the display upon the next display refresh, i.e., the arrival of a VSync signal [14], which is mostly generated in 60 Hz on a mobile device. Fig. 1 illustrates the overall processing flow of an event.

QoS Experience A user's QoS experience is determined by the event execution latency, which is the delay between when an event (interaction) is triggered to when the corresponding frame is visualized on the display. Prior work has shown that mobile users tend to have a maximally tolerable delay of each event, also called the *QoS target* [28, 65, 69]. Going beyond the QoS target, would lead to an unsatisfactory QoS experience. In the example of Fig. 1, the event latency meets the event's specified QoS target. Note that the event latency includes an idle period between when a frame is prepared and when the display refreshes.

3 Experimental Setup

Software Setup We use Google's open-source Chromium Web browser[7] (Version 67.0.3360.0) as the experimental Web runtime. Chromium is the basis of not only the off-the-shelf Chrome browser, but also many Web runtime systems, e.g. the Android WebView [17].

Application Selection We study a suite of 12 mobile Web applications that are previously used in similar studies [72]. These 12 applications are ranked among the Alexa's top 25 webpages [2] and are representative of the top 10,000 webpages in terms of both application-inherent and hardware-dependent features based on principal component analysis.

Hardware Setup We use the ODroid XU+E development board [15] as a representative mobile hardware platform. The ODroid XU+E board contains a Samsung Exynos 5410 SoC that is used in Samsung Galaxy S4 smartphone among other commercial mobile devices. The Exynos 5410 SoC contains an ACMP subsystem, which includes a high-performance, energy-hungry (big) core cluster consisting of four out-of-order Cortex A15 processors and a low-performance, energy-conserving (little) core cluster consisting of four in-order A7 processors. A15 operates between 800 MHz and 1.8 GHz at a step of 100 MHz while A7 operates between 350 MHz and 600 MHz at a step of 50 MHz.

Energy Measurement We focus on the processor energy consumption because the processor has gradually become the most significant power and energy consumer in a mobile device compared to other components such as the display and network. We leverage the build-in current sense resistors on the ODroid board to directly measure the processor power. We use the National Instruments DAQ Unit X-series 6366 to simultaneously collect voltage measurements of both the big and small CPU clusters at 1 KHz.

4 Motivation and Characterizations

This section motivates the idea of proactive scheduling through systematically characterizing mobile Web applications. We first introduce the Web runtime scheduling and describe the inherent reactivity of existing scheduling schemes (Sec. 4.1). We then use a representative example to explain the inefficiencies of reactive scheduling mechanisms (Sec. 4.2), and show that the sources of inefficiencies are prevalent in mobile Web applications in general (Sec. 4.3).

4.1 Web Runtime Scheduling

The runtime scheduler is an important component of Web runtime systems. The scheduler determines how to best execute Web applications on the underlying system in order to optimize for user QoS and energy-efficiency. Conventional schedulers leverage "software knobs" such as deciding the task order or throttling background activities to give more memory resources to foreground tasks [6]. Recent advancements in Web runtime schedulers are increasingly hardware-aware. In particular, this work considers hardware systems that incorporate the ACMP architecture due to their prevalence in today's mobile devices [10, 16]. The ACMP architecture consists of multiple cores with different microarchitectures (e.g., out-of-order and in-order). Each core has a variety of frequency settings. Different core and frequency combinations expose a large latency-energy trade-off space to the runtime scheduler.

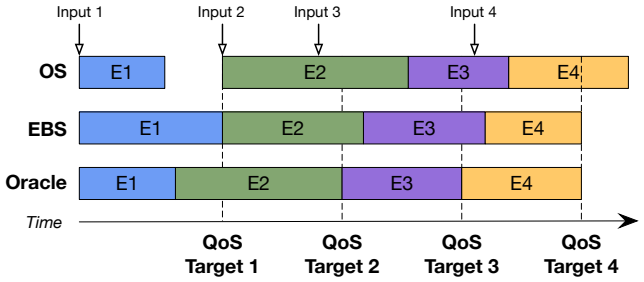


Fig. 2: Comparison of different scheduling mechanisms using a representative interaction sequence from `cnn.com`. Each input (top of the figure) triggers an event execution, and corresponds to a particular QoS target (bottom of the figure). OS and EBS are reactive schedulers and lead to QoS violations or energy waste. The oracle scheduler can proactive coordinate executions across events, and thus eliminates all QoS violations and minimizes energy consumption.

The goal of an ACMP-aware runtime scheduler is to find an ideal ACMP execution configuration (i.e., a <*core*, *frequency*> tuple) such that the events' QoS targets are satisfied with a minimal energy consumption. Switching off cores is not beneficial due to tiny slacks between two events thus not included here. Existing OS schedulers (e.g., the Android CPU governor [3]) are QoS-agnostic in that they do not take into account an event's QoS target during scheduling. Recent work has started investigating event-based, QoS-aware scheduling mechanisms that attempt to minimize energy in the presence of event QoS targets [43, 57, 69].

However, all existing schedulers suffer from one major inefficiency: they are *reactive* by nature as they consider only events that have been triggered. As a result, they necessarily apply *local* scheduling decisions in that they schedule events one at a time without considering the interferences from other events. By event interference, we refer to the fact that the execution of the current event will necessarily affect the start time of the subsequent events. Collectively, existing schedulers lack the ability to coordinate across events and miss optimization opportunities.

We now use one representative mobile Web application as a case-study to explain the inefficiencies of reactive schedulers. We then expand our analysis to include a comprehensive set of applications to show the general trends.

4.2 Representative Analysis

We use a snapshot of an event sequence taken while interacting with `cnn.com` to illustrate the inefficiencies of existing schedulers. The interaction trace contains four inputs, each of which triggers an event execution. Each event has a particular QoS target (deadline) that the runtime system strives to meet in order to achieve responsiveness. We focus on the three fundamental user interactions: load, tap, and move, and use 3 s, 300 ms, and 33 ms as the QoS target for their corresponding events, respectively [73]. We abstract away the event details and use numeric representations to denote the events.

We compare three different scheduling mechanisms in Fig. 2: (1) an OS scheduler using the `Interactive` CPU governor that is QoS-agnostic, (2) EBS which represents the state-of-the-art QoS-aware scheduler, and (3) an oracle scheduler.

OS Scheduler The OS scheduler finishes the first event E1 before the deadline. However, E1 leaves a latency slack that could have been exploited to save energy by lowering the hardware capability. The second event E2 misses the deadline and thus violates the user QoS requirement. This QoS violation happens because the OS scheduler does not explicitly consider QoS targets; instead, it adjusts the ACMP configurations based on the CPU utilization. E2 has a low CPU utilization (< 70%), and is scheduled to a low-performance configuration by the OS, which is insufficient to meet the QoS target. E2's QoS violation delays the processing of E3 and E4, which subsequently also miss their deadlines.

QoS-Aware Scheduler Recognizing the inefficiencies in the OS scheduling, EBS [69] explicitly schedules each event under its QoS target to better optimize for responsiveness and energy-efficiency. Specifically, before executing an event EBS predicts the optimal ACMP configuration that would meet the event's QoS target using the minimal energy. Fig. 2 illustrates the improvement of EBS over the OS scheduler. For instance, EBS exploits the latency slack of E1 and thus saves energy.

However, EBS has limitations. First, EBS misses the deadlines of E2 and E3. It misses E2's deadline because the inherent workload of the E2 is so high that even the most powerful ACMP configuration could not provide enough performance. However, E3 would have met the deadline if scheduled individually, but misses the deadline in EBS. This is due to the interference from E2, which reduces E3's time budget. Second, EBS wastes energy on E4. This is because E4 is delayed due to the interference of E3; EBS meets the QoS target of E4 by scheduling it to a higher-performance configuration. However, if scheduled individually E4 could have met its deadline with lower performance and lower energy consumption.

Overall, by being QoS-aware EBS eliminates the QoS violation of E4 and saves energy for E1. However, its limited scheduling scope of outstanding events only and inability to schedule across events lead to QoS violations for E2 and E3 and wastes energy on E4.

It is worth noting that the QoS violations introduced by EBS (i.e., E2 and E3) are *not* caused by the event queuing delay. We find that the average event queue length is below 2: events almost always do not wait. This is because, different from servers that could experience traffic surges, human naturally generates interactions slowly. Thus, improving the speed of the scheduler itself would have a marginal effect. In fact, EBS has a scheduling latency < 1 ms.

Oracle Scheduler The inefficiencies of EBS stem from its reactive nature. EBS schedules events only after they are triggered, and therefore has a limited scheduling scope. As a comparison, Fig. 2 shows the execution profile of an oracle scheduler that has *a priori* knowledge of the four-event sequence. The oracle scheduler shortens the execution of E1 to leave enough time for E2, which in turns allows for enough time for E3 and E4. In this way, E3 and E4 not only meet the QoS targets, but can achieve so with lower-performance configurations that save energy. Overall, the oracle scheduler meets the QoS targets for all four events and reduces the total energy consumption by almost one-fourth compared to EBS.

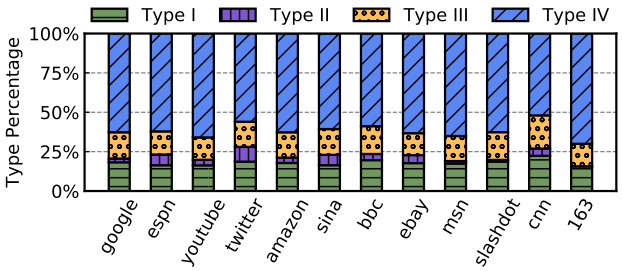


Fig. 3: Distribution of events types under EBS. Type I and Type II events violate QoS. Type III events meet QoS while consuming energy more than necessary. Type IV events are "benign"; they meet QoS and provide opportunities to accommodate other types of events in a proactive scheduling.

Critically, both E2 and E4 start executions before the corresponding inputs are triggered. Such a proactive schedule is fundamentally unattainable in reactive mechanisms such as EBS and the OS scheduler, and indicates the potential of a proactive scheduler that anticipates future events and coordinates event executions globally.

4.3 Comprehensive Analysis

We now expand the analysis to all 12 applications in the benchmark suite, and quantify the prevalence of the reactive schedulers' inefficiencies. To simplify discussion, we categorize events under EBS into four categories. Note that the event categorization is not intrinsic to the events, but depends on where an event appears, which in turn reflects the scheduling policies. Our goal of event categorization is to understand the limitations of different scheduling policies, rather than the intrinsic characteristics of events.

- Type I: Events whose workloads are inherently high such that even the highest-performance hardware configuration does not meet the QoS. The E2 in Fig. 2 is an example of a Type I event. For Type I events, conventional schedulers tend to consume high energy by supplying the highest-performance configuration in order to meet the deadline. However, a proactive scheduler would be able to coordinate it with its preceding events and thus meet the QoS target with lower energy.
- Type II: Events that could meet the deadline with a proper hardware configuration if scheduled individually, but miss the deadline at runtime due the interferences from other events. The E3 in Fig. 2 is an example of a Type II event. A proactive event scheduler would coordinate Type II events with their preceding events and thus meet the QoS targets with lower energy.
- Type III: Events that could meet the deadline if scheduled individually, and do meet the deadline at runtime but require higher performance than necessary due to the interferences from other events. The E4 in Fig. 2 is an example of a Type III event. A proactive event scheduler would coordinate Type III events with the interfering events so as to further exploit latency slacks to save energy.
- Type IV: Events that could meet the deadline with a proper hardware configuration if scheduled individually, and do not encounter interference during runtime and thus meet

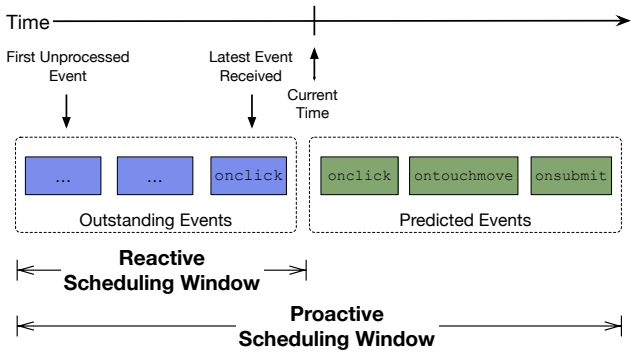


Fig. 4: Our proactive scheme unlocks more optimization opportunities by enlarging the scheduling window through predicting events that will likely happen in the future.

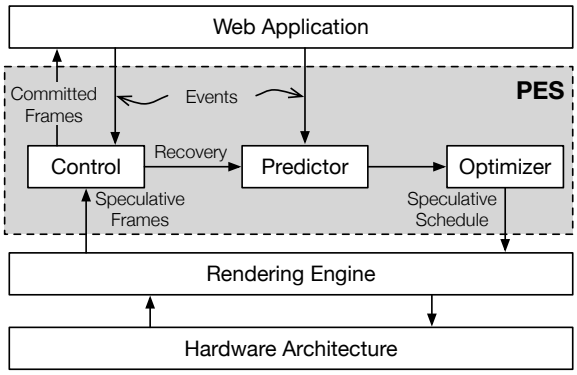


Fig. 5: Overview of an PES-augmented mobile Web stack. PES layer is shaded. See Fig. 6 for the detailed design of PES.

the QoS. The E1 in Fig. 2 is an example of a Type IV event. These events could be leveraged by a proactive scheduler to accommodate events of the previous three types for global QoS/energy optimizations.

Fig. 3 shows how the events are distributed across the four categories. The results reveals two general trends that corroborate the observations made from the representative analysis. First, on average 21% of the events miss the QoS target (i.e. the sum of Type I and II), and 14% of the events potentially waste energy in meeting the QoS target (i.e. Type III). Therefore, a reactive scheduler (e.g, EBS) does not deliver optimal results for 35% of the events, indicating large room for improvement.

Second, the number of Type I events is almost the same as the sum of Type II and Type III events across applications. Our further investigation shows that this is because whenever a Type I event occurs, a Type II or Type III event is mostly likely to follow. The co-occurrences between Type I and Type II/III events indicate that a global scheduler that optimizes across events is likely to perform better than a local scheduler that optimizes individual events alone.

5 Proactive Event Scheduling

This section introduces PES, a proactive event scheduler that addresses the inefficiencies in reactive schedulers. We first provide an overview of the scheduler (Sec. 5.1). We then discuss the detailed PES design, emphasizing its three components: the event predictor (Sec. 5.2), the global optimizer (Sec. 5.3), and the control unit (Sec. 5.4). We provide implementation details in the end (Sec. 5.5).

5.1 Overview

We first give an intuitive illustration of the execution model under PES, and then describe the high-level workflow of PES.

Execution Model Existing mobile Web runtime schedules events in a reactive way in that the scheduler reacts to only outstanding events, i.e., events that users have already generated but not executed yet. The key idea of PES is to proactively anticipate future events and thereby coordinate scheduling decisions globally.

To illustrate this idea, Fig. 4 compares reactive schedulers and the proposed proactive scheduler. Existing reactive schedulers has a scheduling window limited by outstanding events that have already happened and perceived by the application. For instance in a survey section of a mobile application, a user might have triggered an `onclick` event on a checkbox, which, along with other events that have been triggered but not yet served, is scheduled by the scheduler. The scheduler optimizes for responsiveness and energy-efficiency by leveraging the ACMP "knobs".

On the contrary, the proactive runtime predicts what events will likely happen in the immediate future and coordinates scheduling decisions accordingly, as illustrated in Fig. 4. For instance, it is likely that once a user clicks a checkbox, a series of `onclick` and `ontouchmove` events will occur followed by an `onsubmit` to submit a form. With such a prediction capability, the scheduler can take a "sneak peek" of future events. In this way, the proactive event scheduler not only schedule outstanding events that are waiting to be served, but also considers future events that are about to happen. PES thus enlarges the scheduling window and unlocks more optimization opportunities.

Framework Overview In a conventional mobile Web stack, user interactions with applications (events) are forwarded to the rendering engine, which produces frames on the hardware and submits the frames to the application in reaction to the events. PES is built on top of this architecture by adding an additional layer between the rendering engine and the application. Fig. 5 shows an overall architecture of the PES-augmented mobile Web stack.

The PES layer contains three main modules, a *predictor*, an *optimizer*, and a *control unit*. In essence, the predictor predicts a sequence of future events, which along with outstanding events are fed into the optimizer that calculates the optimal schedule, which minimizes the overall energy consumption while satisfying the QoS constraints of each event. The schedule calculated by the optimizer is in a speculative state because the predicted events have not been validated with the actual user inputs. The speculative schedule is then sent to the rendering engine, which in turn executes the schedule on the ACMP hardware to generate speculative frames.

While speculative frames are being generated, the control module monitors the actual user input events. If an actual input event matches a predict event, the controller would commit the corresponding speculated frame to the application for display; otherwise the controller drops all the remaining speculative frames, and

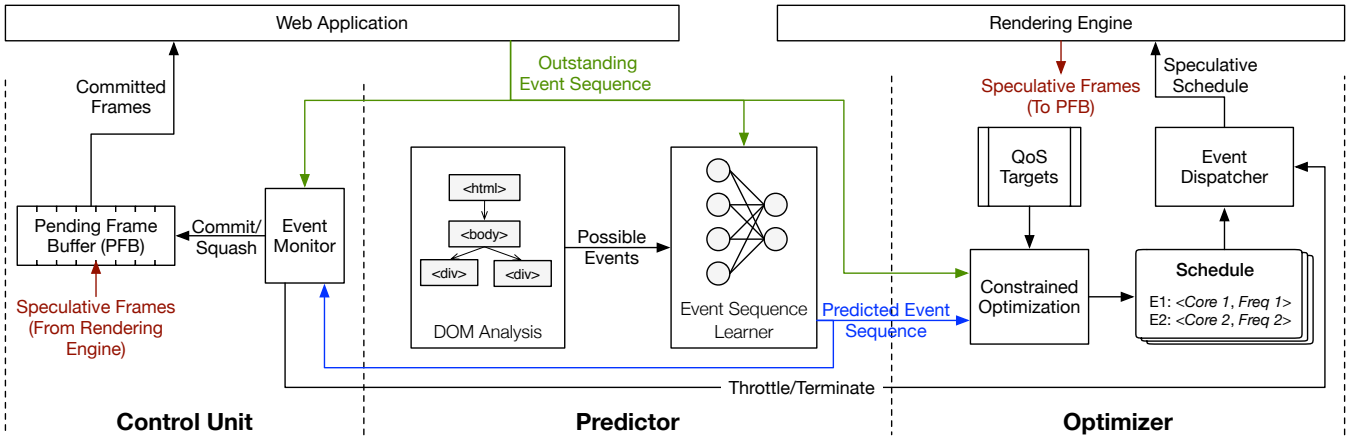


Fig. 6: Detailed PES design. The application, the PES, the rendering engine constitute a feedback-driven optimization loop.

alarms the predictor to reboot prediction. When all the speculative frames are committed, i.e., no predicted events are left, the predictor starts a new around of event sequence prediction.

5.2 Event Predictor

The event predictor predicts the upcoming events during a user interactive session. Fig. 6 shows the detailed design of PES, where the predictor feeds the sequence of predicted events to the optimizer, and interfaces with the controller to handle mis-predictions.

The key idea of predicting future events is to combine statistical inference and program analysis. We find that user interactions constitute an event sequence that exhibits strong temporal behaviors that could be statistically inferred. Meanwhile, application-inherent logics dictate all the possible future events, tightening the prediction space. Fig. 6 shows the interaction of these two.

Event Sequence Learner Formally, the goal of the event sequence learner is to estimate the following conditional probability: $p(y_1, ..., y_{T'}|x_1, ..., x_T)$, where $\{x_1, ..., x_T\}$ is the event sequence occurred so far, and $\{y_1, ..., y_{T'}\}$ is the predicted sequence. The event sequence learner operates in a recurrent fashion where every step generates a feature vector to predict the immediate next event. The predicted event is fed back to the learner to predict the subsequent event. Note that, the event sequence learner only predicts the type of immediate next event, not when it will be triggered.

We choose to construct the prediction model based on logistic regression. Specifically, the event sequence learner employs a set of logistic models, each of which estimates the probability of one possible next event through $ln(p/(1-p)) = \mathbf{x}\beta$, where p is the probability that an event will be triggered, $\mathbf{x}$ is the feature vector, and β are trained coefficients. In the end, the event with the highest probability is deemed the next event. There are other alternatives for temporal prediction such as Long Short-Term Memory (LSTM). However, we find that logistic regression provides sufficient accuracy with low compute overhead as we will quantify in Sec. 6.3.

The predictor must adapt to different application contents and user behaviors. Therefore, we propose to construct the feature vector by considering both application-inherent information and runtime information of the current interaction sequence. Table 1 lists the specific features that we consider. In particular, we consider the runtime information within a window of the five most recent events. The combination of application-inherent and interaction-dependent features is aimed to adapt to different users and applications.

Each event prediction is associated with a confidence value denoted by the output of the logistic model, i.e., p. If the *cumulative* confidence of the event sequence (i.e., the product of the confidence values of all the predicted events in the sequence) is below a threshold, the event sequence learner terminate the prediction and sends the predicted event sequence to the optimizer. The confidence threshold is a critical parameter that determines the number of consecutive events the predictor predicts ahead, which we dub *prediction degree*. Intuitively, a greater prediction degree increases the scheduling window but introduces a higher chance of misprediction, and vice versa. We will show in Sec. 6.5 that PES is largely robust against different confidence thresholds. We empirically use 70% in our design.

Web Application Analysis The goal of the program analysis is to identify a set of events that could possibly be triggered, and thus narrow down the prediction space by the event sequence learner. For instance, if a button exists in an application but is not visible on the display (i.e., outside the viewport), the next user input will not trigger an `onclick` event on the button; similarly, no event will be triggered on an image if the image has no event associated with it.

We particularly focus on analyzing the DOM tree of a Web application. The DOM tree is a tree-like representation of the Web application where each node represents an application element. For instance, a submit button likely corresponds to a `button` node on the DOM tree. Each DOM node is registered with a set of events. Our DOM analyzer traverses the part of the DOM tree that is within

Table 1: Model features.

Category	Feature
Application-inherent	Clickable region percentage in the viewport
	Visible link percentage in the viewport
Interaction-dependent	Distance to the previous click in the window
	Number of navigations in the window
	Number of scrolls in the window

```
1 button.addEventListener("click", function() {
2   var content = this.nextElementSibling;
3   if (content.style.display === "block") {
4     content.style.display = "none";
5   } else {
6     content.style.display = "block";
7   }
8 });
```

Fig. 7: Code snippet that toggles a collapsible menu.

the current viewport, and accumulates a set of events that are associated with the visible DOM nodes, which we call the *Likely-Next-Event-Set* (LNES). The event sequencer learner would then predict the next event out of LNES.

The challenge of the DOM analyzer is that one event's execution might mutate the visible part of the DOM tree, and thus makes identifying the LNES for the next event difficult. Addressing this challenge is critical to enable the event sequence learner to predict multiple events consecutively so as to enlarge the scheduling space.

As a specific example, triggering an `onclick` event that expands a menu would show the menu and thus present more visible DOM nodes (i.e., menu items), on which more events could be triggered. Fig. 7 shows a code snippet that toggles a collapsible menu where `content` is the DOM node that corresponds to the menu. The new DOM state after the `onclick` event is not immediately clear as the callback function simply sets the display style from `none` to `block`. Fully evaluating the callback function to follow the `content` DOM node would be a possible solution, but it defeats the purpose of globally scheduling multiple events together.

Instead, we construct a Semantic Tree during parsing, where we memoize that `content` is a button that toggles a menu as well as the fact that the DOM node associated with the menu itself. In this way, the DOM analyzer could statically examine the Semantic Tree to identify the DOM state after the callback without having to dynamically evaluate the callback. We will show an efficient implementation of this design in Sec. 5.5.

5.3 Energy and QoS Optimizer

Upon receiving the predicted event sequence from the predictor, the optimizer computes a speculative schedule by combining the outstanding events with the predicted event sequence. The event dispatcher then issues the schedule to the rendering engine, which executes each event according to the schedule. We now describe schedule computation and the event dispatcher. Mis-predictions are handled by the control unit, which will be discussed in Sec. 5.4.

Optimization Intuitively, the optimization component determines the ACMP configuration (a <*core*, *frequency*> tuple) for each event in a way that the total energy consumption across all the scheduled events is minimized while the latency deadline (QoS target) of each event is met. We find that the scheduling task can be formulated as a constrained optimization problem, which can be efficiently solved by integer linear programming (ILP). We now describe our formulation.

We leverage the classical DVFS analytical model [64] that estimates the execution time T of a code segment:

$$T = T_{mem} + N_{dep}/f \tag{1}$$

where T_{mem} is the time for accessing the memory, f denotes the CPU frequency, and N_{dep} is the number of CPU cycles that are not overlapped with the memory access. For the first two times an event is encountered, we measure its latency under two different frequencies and solve the system of equations as formulated by Eqn. 1 to obtain the values of T_{mem} and N_{dep}. This is well-established practice used in prior work [43, 69, 73].

Each event i can be scheduled to one, and only one, of the C ACMP configurations. Therefore, the following ACMP configuration constraint is enforced on each event i:

$$\sum_{j=0}^{C} \tau^{(i,j)} = 1,\ \tau^{(i,j)} \in \{0,1\} \tag{2}$$

where $\tau^{(i,j)}$ is a binary value denoting whether a particular ACMP configuration j is assigned to event i. $\tau^{(i,j)}$ is 1 only when the configuration j is active when executing event i.

Combining Eqn. 1 and Eqn. 2, the event latency of an event i, denoted as $\Delta t^{(i)}$, is modeled as:

$$\Delta t^{(i)} = T_{mem} + \sum_{j=0}^{C} N_{dep}/f^{(j)} \times \tau^{(i,j)},\ \tau^{(i,j)} \in \{0,1\} \tag{3}$$

where f^j is the frequency under the configuration j.

In order to meet the QoS target, our formulation imposes a deadline, $t_c^{(i)}$, for every event i. That is:

$$t^{(i-1)} + \Delta t^{(i)} \leq t_c^{(i)}, \forall\ i \in \{0, ..., N-1\} \tag{4}$$

where $t^{(i-1)}$ is the end time of the previous event's execution, and $\Delta t^{(i)}$ is the latency of the current event i.

The objective of the scheduler is to minimize energy consumption, which we model based on the event latency model (Eqn. 3) and a power model. We construct the power model as a look-up table because the hardware exposes only a limited number of discrete frequencies. We measure the power consumption of all the frequency and core combinations offline, and persist them in a local storage file, which gets loaded into the runtime by PES when an application boots. This is similar to the practice in prior work [69]. Note that the energy consumptions in the evaluation are *measured* rather than using this power estimation model.

Given the constraints and the power modeling, we formulate the scheduling task as an optimization problem:

$$\begin{aligned} &min \sum_{i=0}^{N} p^{(i)} \times \Delta t^{(i)} \\ &s.t.\ t^{(i-1)} + \Delta t^{(i)} \leq t_c^{(i)}, \forall\ i \in \{0, ..., N-1\} \end{aligned} \tag{5}$$

where $p^{(i)}$ is the power consumption of event i, and N is the total number of scheduled events. This optimization problem is formulated with respect to the variables $\tau^{(i,j)}$ (Eqn. 2), and both the constraints and the objective are linear with respect to $\tau^{(i,j)}$.

Event Dispatcher Solving the optimization problem in Eqn. 5 generates a speculative schedule that assigns an ACMP configuration to each event. The event dispatcher sets up the hardware for each event based on the schedule, and then sends the event to the rendering engine. The event dispatcher would stop dispatching upon receiving a mis-prediction signal from the control unit.

One practical design decision that we take is to suppress issuing network requests before an event is confirmed to be correctly predicted. This is because network requests could have irreversible side effects (e.g., modifying server states).

5.4 Control Unit

The control unit in PES is responsible for validating the event prediction results from the predictor and for handling mis-predictions properly. To that end, the control unit uses a Pending Frame Buffer (PFB) to hold all the speculative frames generated from the speculative schedule. If a predicted event is confirmed to match an actual event that has occurred, the event monitor in the control unit signals the PFB to commit the corresponding speculative frame to the application for display. When all the frames are committed, the controller informs the predictor to start a new around of prediction to generate a new sequence of predicted events.

Handling Mis-predictions Mis-predictions are rare as we will quantify in Sec. 6.2, and are very lightweight to handle when they occur. Upon an event mis-prediction, the controller simply drops all the speculative frames in the PFB, terminates the event dispatcher, and informs the event predictor to re-start the prediction. Note that the work spent on generating the frames for mis-predicted events is wasted, but the waste is minimal as we will quantify in Sec. 6.3.

In addition, the control unit will disable prediction altogether if it experiences multiple (> 3) mis-predictions in a row. In that case, PES falls back to use the best reactive scheduler (i.e., EBS in our paper). This design decision allows PES to be robust against unexpected event behaviors.

5.5 Implementation Details

Constrained Optimization The optimization problem formulated in Eqn. 5 can be solved by integer linear programming. We implement our own solver customized to this particular formulation instead of using a thirty-party solver such as GLOP [9] in order to improve the runtime efficiency.

Constructing the Semantic Tree We piggyback the implementation of the Semantic Tree on top of the Accessibility Tree (AT) [8] that is widely supported in all major Web browsers such as Chrome [1] and Firefox [13].

The AT is similar to the DOM tree in structure, and reflects the semantics attributes of all the accessible nodes in the DOM tree. For instance, an AT node would tell us whether a `<div>` node is a clickable button or just a piece of text, and when click the button which other `<div>` node will become dropdown menu. In this way, by inspecting the AT the DOM analyzer could easily identify the DOM state after an event is triggered (i.e., the menu is expanded). Extending the AT for the Semantic Tree adds little runtime and implementation overhead.

Predictor Training To construct the event prediction model, we record over 100 interaction traces from different users [33] for the 12 applications that we study (Sec. 3). The traces faithfully record the timing of each event, including the user pause (thinking) time. On average, each interaction trace lasts about 110 seconds and contains about 25 total number of events (up to 70). These statistics are consistent with prior user studies in mobile computing [34]. Our traces cover three primitive user interactions in mobile Web: loading, tapping, and moving [73], and include different manifestations of the same interaction. For instance, our traces contain both `click` and `touchstart` events for the same "tapping" interaction.

The event sequence model is trained using training traces from all applications so as to be generally applicable to different applications. However, the DOM analysis at runtime naturally guides the predictor to be application-specific. We train the model offline. Training takes as little as 3 seconds on an Intel Core i5-7500 CPU at 3.40GHz, indicating the convenience of re-training if necessary. The predictive model is then integrated into Chromium Web runtime.

6 Evaluation

We first describe our evaluation methodology (Sec. 6.1). We then quantitatively show that the event predictor achieves high prediction accuracies, generalizes well to unseen applications (Sec. 6.2), and introduces negligible overhead (Sec. 6.3). As a result, PES outperforms both the Android default mechanisms as well as the state-of-the-art reactive scheduler (Sec. 6.4). Finally, we conduct a sensitivity analysis to show the robustness of PES (Sec. 6.5).

6.1 Evaluation Methodology

Evaluation Benchmark We evaluate PES using 18 applications, which include the 12 applications used in Sec. 3 as well as six unseen applications in order to understand the generalizability of PES. We collect three traces for each application, and each trace is replayed under different scheduling mechanisms [33]. Note that all the evaluation traces are different from the training traces used in Sec. 5.5 regardless of whether the applications are seen before or not. That is, we collect *new* user traces for evaluation.

Metrics We use two metrics to evaluate PES: QoS violation reduction and energy savings. We define a QoS violation as an event's execution that exceeds a specified QoS target (deadline). We report the average energy consumption and QoS violation across all the events in an application.

Baseline We compare against three baseline mechanisms:

- `Interactive`: This is the Android's `Interactive` scheduler designed specifically to enable better interactivity. It is the default CPU governor [3]. It periodically samples the CPU utilization, and maximizes the CPU frequency if the CPU utilization is above 85%.
- EBS: This is the Event-based Scheduler that represents a class of reactive schedulers that optimize event executions according to their QoS targets. Before executing an event, EBS predicts the optimal ACMP configuration that meets the event's QoS target using the minimal energy. We implement EBS as described in Zhu et al. [69].
- `Oracle`: This is the oracle scheduler that has *a priori* knowledge of the entire event sequence. It maximizes the energy savings during the entire application lifetime while minimizing the QoS violations.

6.2 Event Predictor Accuracy

Fig. 8 shows the predictor's accuracies across the 18 applications. The accuracy is the defined as the percentage of correctly predicted

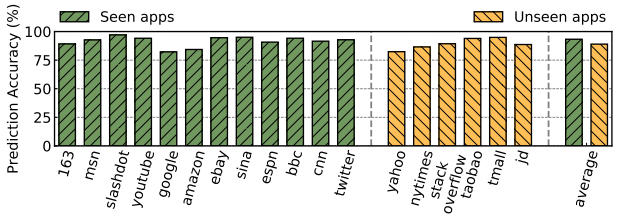


Fig. 8: The event predictor accuracy. Note that all the evaluation traces are collected from new users regardless of whether the applications are seen before or not.

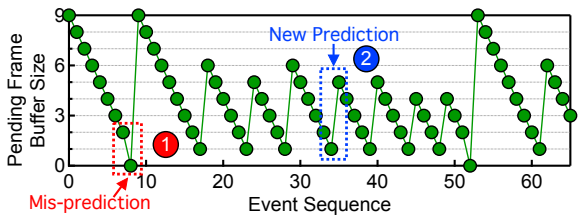


Fig. 9: Pending frame buffer (PFB) size changes over time. We highlight one mis-prediction instance and one new prediction instance. Other instances are omitted due to space.

events. Our predictor achieves a 91.3% prediction accuracy on average (4.1% standard deviation) for the user interactions in the 12 seen applications. The high prediction accuracies indicate the feasibility of a simple logistic regression-based prediction model. Our predictor generalizes well to unseen applications, achieving an 89.2% prediction accuracy (4.7% standard deviation) for the six unseen applications. The generalizability of the predictor is a direct result of the design that augments a generic event sequence learner with application-specific DOM analyses (Sec. 5.2).

Using ebay as a case-study, Fig. 9 illustrates the dynamics of event prediction, where each <x, y> marker represents the number of speculative frames in the Pending Frame Buffer (PFB) (y) when a new event occurs (x). As described in Sec. 5.4, when a new event occurs and is matched with a predicted event, the corresponding speculative frame will be committed and the PFB size gets decremented by 1. This is common during the application execution.

Upon a mis-prediction, all the frames in the PFB are dropped and the PFB size drops to 0, as is the case of ❶. When the last predicted event is matched and the speculative frame is committed, the predictor starts a new round of prediction, and the rendering engine pushes a new set of speculative frames into PFB, as case ❷.

We also observe that the prediction accuracy varies across different applications. Specifically, the accuracy varies from 97.0% (slashdot) to 82.2% (google). Further investigations show that the accuracy variance is mainly affected by two factors: the intrinsic properties of the application (e.g., event and DOM tree characteristics) and user interactions with the application. For instance, the prediction on applications with larger clickable area, such as amazon, is generally harder to predict compared to applications with less clickable applications such as slashdot. This indicates that an application-specific event sequence learner can potentially further improve prediction accuracy, which we leave as future work.

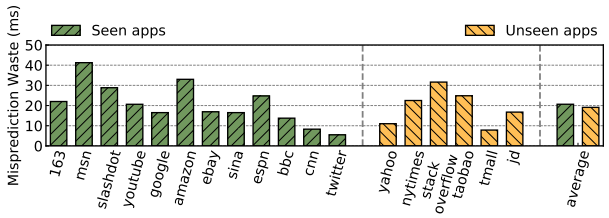


Fig. 10: The average mis-prediction waste.

6.3 Overhead Analysis

Runtime Overhead PES introduces three sources of addition work, all of which are negligible and are far out-weighted by the benefits of PES. First, predicting the user events involves evaluating a simple five-variable logistic model with an overhead of about 2 μs. Second, solving the constrained optimization problem takes about 10 ms, which itself is amortized across multiple event executions. Finally, switching CPU frequencies and core migration incurs an overhead of 100 μs and 20 μs, respectively [69, 71]. The overheads are negligible compared to typical event latencies that range from hundreds of milliseconds to several seconds, and are captured by the real-system measurements.

Mis-prediction Waste Although handling mis-prediction has almost zero cost because it involves only flushing the speculative frames, mis-predictions waste the work spent on generating the speculative frames. We define *mis-prediction waste* as the time it takes to generate a speculative frame (which is eventually discarded) for a mis-predicted event. Fig. 10 shows the average mis-prediction waste across different applications. For both seen and unseen applications, the average mis-prediction waste is about 20 ms, which translates to an amortized waste of 2 ms per event. The average energy overhead introduced by a mis-prediction is 7.2 mJ (1.8%) and 8.2 mJ (2.2%) for seen and unseen applications, respectively. Combining the high prediction accuracy and the low mis-prediction waste, PES achieves significant energy savings as we show next.

6.4 Energy and QoS Evaluations

Energy Saving We now compare PES with the three baseline systems described in Sec. 6.1. Fig. 11 shows the energy consumption of the four schemes normalized to Interactive for both seen applications and unseen applications. We find that Interactive spends over 80% of the time running on the big core using the highest frequency, and thus consumes the highest energy. In comparison, EBS is aware of events' QoS targets and thus is able to select a proper ACMP configuration for each event that meets the QoS target using minimal energy. On average, EBS is able to achieve about 10.2% energy savings compared to Interactive.

However, EBS is limited by its reactive nature that schedules only events that have occurred. In contrast, PES predicts future events to enlarge the scheduling window, and coordinates executions across events. For the 12 seen applications, PES reduces the energy by 27.9% and 19.8% compared to Interactive and EBS, respectively.

PES is within 12.9% of the energy savings of Oracle. The gap between PES and Oracle mainly comes from two sources. First, Oracle can predict infinitely far ahead (i.e., an infinite *prediction degree*) because it has the knowledge of the entire event sequence,

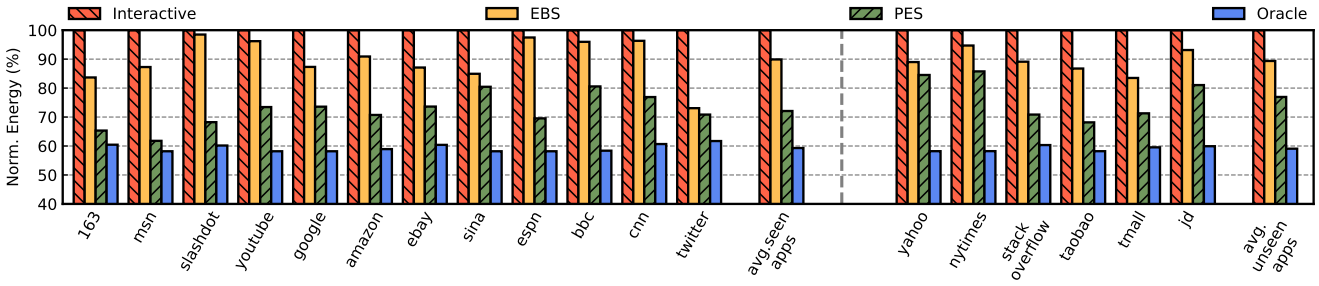


Fig. 11: Energy consumption normalized to `Interactive`, which consumes the highest energy among all four schemes. Lower is better. The results of the 12 seen applications are on the right, and the six unseen applications are on the left.

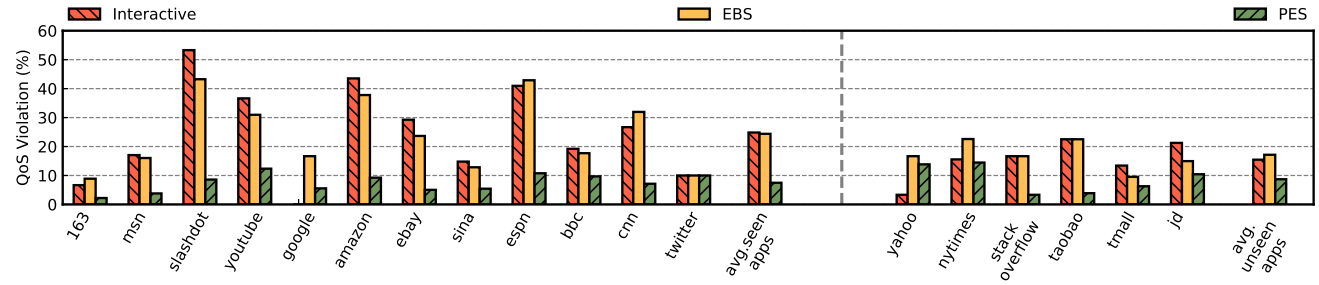


Fig. 12: QoS violation. Lower is better. The `Oracle` scheduler has zero QoS violation and thus not shown. The results of 12 seen applications are shown on the right, and the results of the six unseen applications are on the left.

whereas PES predicts only about five events ahead (i.e., a *prediction degree* of five) before the prediction confidence drops below the threshold (See Sec. 5.2). Second, `Oracle` also has a perfect prediction accuracy while PES incurs mis-predictions as shown in Fig. 8.

Interestingly, a high prediction accuracy does not always lead to a high energy saving (e.g., `sina`). This is because the prediction accuracy is not the only factor affecting energy saving. The computation intensity of the events matters too. We find that `sina` contains many compute-light events; scheduling them to a low-performance configuration leads to lower energy savings than applying the same scheduling to compute-intensive events.

To show the generazability of PES, we also evaluate PES under the six unseen applications. The results are shown in Fig. 11. On average, PES achieves 23.1% and 13.9% energy savings compared to `Interactive` and EBS, respectively, both of which are slightly lower than the savings obtained from seen applications because of the slightly slower event prediction accuracy as shown in Fig. 8.

QoS Violation Fig. 12 shows the QoS violations across the four schemes. `Oracle` completely removes the QoS violation for all the applications (and thus not shown) because it can schedule across the entire event sequence. Across the 12 seen applications, `Interactive` and EBS incur a QoS violation at about 24.8% and 24.4%, respectively. In contrast, PES decreases the QoS violation to below 10% for most applications. On average, PES reduces the QoS violation to only 7.5%, indicating that PES can simultaneously improve QoS and reduce energy compared to existing schedulers.

We also show the QoS violation reduction across the six unseen applications in Fig. 12. PES reduce 43.7% and 49.2% of the QoS violations incurred in `Interactive` and EBS.

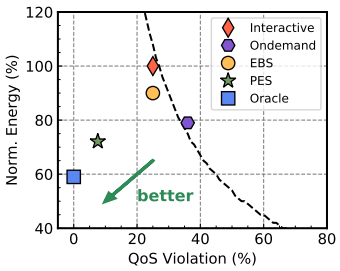
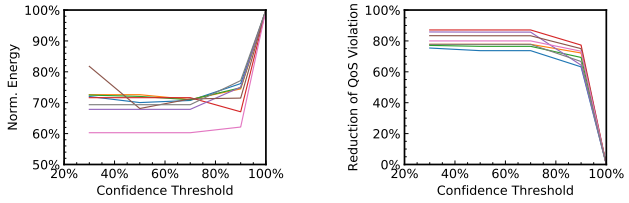


Fig. 13: Pareto analysis of different scheduling mechanisms. Energy values are normalized to `Interactive`. PES Pareto-dominates existing schemes.

Pareto Analysis To summarize the benefits of PES, Fig. 13 shows the Pareto-optimal frontier of all existing schemes. For the sake of completeness, we also show the results of the Android's `Ondemand` CPU governor, which favors energy savings and has much greater QoS violations, and thus is rarely used in interactive applications. PES achieves lower energy consumption even compared to `Ondemand`. Overall, PES Pareto-dominates all schemes.

6.5 Sensitivity Study

Predictive Degree We study the sensitivity of PES with respect to one key parameter of the event predictor: the *prediction degree*. Intuitively, a greater prediction degree provides more opportunities for cross-event optimizations but also introduces higher chances of mis-predictions that degrade efficiency; vice versa.



(a) Normalized energy consumption. Lower is better.

(b) Reduction of QoS violation. Higher is better.

Fig. 14: Sensitivity of PES with respect to the confidence threshold. The data is normalized to EBS. Different curves represent different applications.

We control the prediction degree using the confidence threshold. Recall from Sec. 5.2 that the event sequence leaner terminates prediction if the cumulative confidence of the predicted event sequence (the product of all the predicted events' confidence values) is below a threshold. We vary the confidence threshold from 30% to 100%. 100% is a confidence threshold that is too restrictive, under which predictor usually fails to make predictions and falls back to EBS.

Fig. 14 shows how the energy consumptions and QoS violation reductions change with the confidence threshold. The data is normalized to EBS. The energy saving and QoS reduction initially improve as the threshold decreases from 100% to 70%. This is because a relaxed confidence threshold allows for more aggressive predictions, which enlarge the scheduling window. As the threshold further relaxes from 70% to 30%, the energy consumption and QoS improvements are stable because the benefits of having a large scheduling window are offset by the mis-prediction penalties. This analysis suggests that PES is largely robust against the confidence threshold. We empirically choose 70% as described in Sec. 5.2.

Predictor Design PES predicts future events using a combination of statistical inference (event sequence learner) and DOM analysis (Sec. 5.2). PES could work with just the event sequence learner without DOM analysis, but not the other way around because DOM analysis simply identifies the current DOM state without making predictions. On average, we find that the accuracy of predicting future events without the DOM analysis drops by about 5%.

Other Devices While we primarily evaluated PES on the Exynos 5410 SoC, which was released in 2013, we find that PES has similar improvements on other (more recent) mobile devices. We repeated the same experiments on the Parker SoC on Nvidia's recent TX2 board, which was released in 2017. Leveraging the DVFS capability of the Cortex A57 processors in the SoC, PES achieves about 24.6% energy savings compared to `Interactive`. As 75% of today's smartphones use CPUs that are released before 2013 [63], it is important that PES achieves improvements on a variety of mobile devices.

7 Related Work

ACMP Scheduling As the underlying mobile hardware starts embracing the ACMP heterogeneous architecture, the runtime scheduler has also become ACMP-aware. Traditional ACMP schedulers such as OS governors [3, 52] are QoS-agnostic, and thus tend to miss event/job QoS targets or waste energy. Recent work has started investigating QoS-aware schedulers that make scheduling decisions based on individual event's QoS targets [21, 39, 43, 55, 57, 69]. Most of such scheduling techniques are based on predicting the execution latency of the next schedulable event using history information [21, 69], machine learning [55, 57, 71], and a combination of profiling and machine learning [43]. Others have studied applying control theory to ACMP scheduling [32, 39**?**].

PES has two key distinctions. First, all existing schedulers schedule only events that have been triggered while PES is the first work that predicts the occurrence of future events, and speculatively executes future events. Second, all existing schedulers schedule events one at a time while PES co-schedules outstanding events with future events to enable global QoS/energy optimizations. PES formulates the global scheduling as an ILP problem.

Speculation in Interactive Applications Prior work in mobile computing exploits speculation for various systems optimizations. Outatime [41] speculates user behaviors to improve the interactivity of mobile cloud games. Zare et al. [66] and Haynes et al. [35] propose to predict user head movement to improve network bandwidth-efficiency in VR video streaming. Corm [47] speculative executes JavaScript event callbacks in order to improve the webpage loading performance. In contrast to all prior work, we propose a rigorous optimization framework that co-optimizes performance (responsiveness) and energy-efficiency at the same time.

Energy Optimizations in Mobile (Web) Computing Recent work has refined the definition of QoS in mobile (Web) computing, and proposes new energy optimizations under the new QoS formulations. Gaudette et al., examines the effect of probabilistic QoS which treats QoS guarantee has a probabilistic, rather than deterministic, objective subject to uncertainties in mobile systems [30, 31]. Yan et al., considers QoS variances across different users [65]. Our proactive scheduling mechanism is amenable to both formulations.

PES in its current design is transparent to application developers in that it automatically optimizes for responsiveness and energy-efficiency. Prior work has demonstrated the benefits of empowering developers to better guide runtime optimizations through type systems [22, 26, 56] and program annotations [20, 73]. Future work could investigate language extensions such as hints for predicting future events that could better guide PES scheduling.

Other work has studied improving the energy-efficiency of mobile (Web) computing via hardware augmentations, which are orthogonal to our software-level runtime work. WebCore [72] provides specialized hardware structures for key computation kernels such as CSS style resolutions and key data structures such as the DOM tree. ESP [24] and EFetch [23] augment the hardware for efficient event processing in Web applications. Nachiappan et al. expand beyond CPU and consider SoC-level augmentations to improve the energy efficiency of mobile (Web) applications [48, 49].

Finally, a number of work has focused on reducing the mobile Web energy optimizations through optimizing the display [27, 36, 68] and radio [53, 54]. Instead, PES focuses on optimizing the computation aspect, which has gradually dominated the energy consumption of mobile devices as radio and display technologies improve while the processor architecture becomes more power-hungry [34, 38, 70]. In our measurement of the Samsung Galaxy S4 smartphone that contains the Exynos 5410 SoC, the processor power is about 58% of the total device power under the Interactive

governor while running mobile Web applications. We thus expect that PES will lead to around 17% total device energy reduction.

General Mobile Web Optimization Prior work on mobile Web has primarily focused on improving the absolute performance through paralleling browser tasks [19, 46], smart browser caching [67], resource loading [21, 44], and improving the JavaScript engine [29, 45]. Our work on proactive scheduling focus on user-perceivable performance (i.e., QoS), but can benefit from the absolute performance improvement techniques to better exploit time slack for global optimizations.

Another line of mobile Web research co-optimizes the mobile client with the server, either through directly augmenting the Web server or through a Web proxy [50, 51, 61]. These techniques also exclusively focus on the loading phase of Web applications. PES is a client-only solution that requires no modification to the mobile Web infrastructure, and optimizes the entire application usage session, including the loading phase as well as the post-loading interactions.

8 Discussion

General Applicability While this paper focuses on one particular domain of event-driven applications, i.e., the mobile Web, the fundamental idea of proactive event scheduling is applicable to all event-driven applications. We see a broad applicability of PES to many existing and emerging domains such as cloud services [74] and sensor-rich IoT systems [18] that are all based on the event-driven processing paradigm.

Other Scheduling Knobs This paper specifically focuses on the ACMP architecture because it naturally provides a large scheduling space for performance-energy trade-offs. However, the fundamental idea of proactive scheduling is independent of the scheduling knobs, and could be applied to improve other scheduling mechanisms. For instance, dynamic display resolution scaling [27, 36] and brightness scaling [58] could be better scheduled to accommodate future events that require high user attentions.

Multi-application Environment PES is applicable to multi-programming environment where multiple applications run simultaneously. The reason is two fold. First, today's ACMP architectures provide ample hardware resources, e.g., four big and four small cores in the Exynos 5410 SoC. Therefore, PES still has a large scheduling space even if a background application is occupying some hardware resources. Second, PES is compatible with and can be integrated into recent proposals on interference-aware scheduling [57] by extending the event predictor to consider interference-sensitive features such as core utilization of co-scheduled tasks.

9 Conclusion

To sustain the continuing growth of mobile computing, future mobile systems must compute faster and last longer. Today's mobile systems, however, are limited by their fundamental reactive nature. Reactive systems provision hardware resources to applications as demands arise, and thus are forced to apply localized optimizations due to the myopic view of the system's current states (events). This paper promotes a proactive mobile computing platform that continuously anticipates future application events, and thereby coordinates hardware resources globally across events. We demonstrate the feasibility and benefits of a proactive system in the domain of mobile Web computing through the prototype of PES, which simultaneously improves the energy-efficiency and the responsiveness of mobile Web applications.

References

[1] "The accessibility tree in chrome." https://developers.google.com/web/fundamentals/accessibility/semantics-builtin/the-accessibility-tree
[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] "August 2018 web server survey." https://news.netcraft.com/archives/2018/08/24/august-2018-web-server-survey.html
[5] "Battery statistics." http://batteryuniversity.com/learn/archive/battery_statistics
[6] "Blink Scheduler." https://docs.google.com/document/d/11N2WTV3M0IkZ-kQlKWlBcwkOkKTCuLXGVNylK5E2zvc/edit
[7] "Chromium browser." http://www.chromium.org/Home
[8] "Core accessibility api mappings 1.1." https://www.w3.org/TR/core-aam-1.1/
[9] "The glop linear solver." https://developers.google.com/optimization/lp/glop
[10] "Heterogeneous multi-processing solution of exynos 5 octa with arm big.little technology."
[11] "How 1s could cost amazon $1.6 billion in sales." https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales
[12] "iOS Developer Library: UIWebView." https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/
[13] "Mozilla accessibility architecture." https://developer.mozilla.org/en-US/docs/Mozilla/Accessibility/Accessibility_architecture
[14] "Nvidia: Adaptive vsync technology." http://www.geforce.com/hardware/technology/adaptive-vsync/technology
[15] "Ordroid xu+e development board." http://hardkernel.com/main/products/prdt_info.php?g_code=G137463363079
[16] "Qualcomm unveils kryo 385: Semi-custom a75 and a55 cores, 30% better performance." https://www.xda-developers.com/qualcomm-snapdragon-835-kryo-385-cpu-cores/
[17] "WebView for Android," 2014. https://developer.chrome.com/multidevice/webview/overview
[18] S. Alam, M. M. Chowdhury, and J. Noll, "Senaas: An event-driven sensor virtualization approach for internet of things cloud," in Proceedings of the 1st IEEE International Conference on Networked Embedded Systems for Enterprise Applications, 2010.
[19] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum, "Towards parallelizing the layout engine of firefox," in Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism, 2010.
[20] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2010.
[21] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao, "Rethinking energy-performance trade-off in mobile web page loading," in Proceedings of the 21st ACM Annual International Conference on Mobile Computing and Networking, 2015.
[22] A. Canino and Y. D. Liu, "Proactive and adaptive energy-aware programming with mixed typechecking," Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017.
[23] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven webapplications," in Proceedings of the 23rd ACM International Conference on Parallel Architectures and Compilation, 2014.
[24] ——, "Accelerating asynchronous programs through event sneak peek," in Proceedings of the 42th ACM/IEEE International Symposium on Computer Architecture, 2015.
[25] D. Chaffey, "Mobile marketing statistics compilation," Apr 2018. https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/
[26] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," in Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2012.
[27] M. Dong and L. Zhong, "Chameleon: a color-adaptive web browser for mobile oled displays," in Proceedings of the 9th ACM International Conference on Mobile Systems, Applications, and Services, 2011.
[28] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer, "Using latency to evaluate interactive system performance," in Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, 1996.
[29] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff et al., "Trace-based just-in-time

type specialization for dynamic languages," in Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009.
[30] B. Gaudette, C.-J. Wu, and S. Vrudhula, "Improving smartphone user experience by balancing performance and energy with probabilistic qos guarantee," in Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture, 2016.
[31] ——, "Optimizing user satisfaction of mobile workloads subject to various sources of uncertainties," IEEE Transactions on Mobile Computing, 2018.
[32] Y. Gu and S. Chakraborty, "Control theory-based dvs for interactive 3d games," in Proceedings of the 45th IEEE Annual Design Automation Conference, 2008.
[33] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in Proceedings of the 14th IEEE International Symposium on Performance Analysis of Systems and Software, 2015.
[34] 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 Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture, 2016.
[35] B. Haynes, A. Minyaylov, M. Balazinska, L. Ceze, and A. Cheung, "Visualcloud demonstration: A dbms for virtual reality," in Proceedings of 9th the ACM International Conference on Management of Data, 2017.
[36] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in Proceedings of the 21st ACM Annual International Conference on Mobile Computing and Networking, 2015.
[37] J. L. Hennessy and D. A. Patterson, Computer architecture: a quantitative approach, Sixth Edition. Elsevier, 2018.
[38] 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 Proceedings of the 10th ACM International Conference on Mobile Systems, Applications, and Services, 2012.
[39] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "Poet: a portable approach to minimizing energy under soft real-time constraints," in Proceedings of the 21st IEEE Real-time & Embedded Technology & Applications Symposium, 2015.
[40] D. Kaeli and P.-C. Yew, Speculative Execution in High Performance Computer Architectures. CRC Press, 2005, vol. 6.
[41] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in Proceedings of the 13th ACM Annual International Conference on Mobile Systems, Applications, and Services, 2015.
[42] P. Lewis, "Rendering performance," 2014. https://developers.google.com/web/fundamentals/performance/rendering/
[43] D. Lo, T. Song, and G. E. Suh, "Prediction-guided performance-energy trade-off for interactive applications," in Proceedings of the 48th IEEE International Symposium on Microarchitecture, 2015.
[44] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "Pocketweb: instant web browsing for mobile devices," in Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2012.
[45] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in Proceedings of the 7th IEEE International Symposium on High Performance Computer Architecture, 2011.
[46] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in Proceedings of the 19th ACM International Conference on World Wide Web, 2010.
[47] J. W. Mickens, J. Elson, J. Howell, and J. R. Lorch, "Crom: Faster web browsing using speculative execution." in Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, 2010.
[48] 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 Proceedings of the 21st International Symposium on High Performance Computer Architecture, 2015.
[49] 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 Proceedings of the 43th ACM/IEEE International Symposium on Computer Architecture, 2016.
[50] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster page loads using fine-grained dependency tracking." in Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, 2016.
[51] R. Netravali and J. Mickens, "Prophecy: Accelerating mobile page loads using final-state write logs," in Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, 2018.
[52] A. S. V. Palladi and A. Starikovskiy, "The ondemand governor: past, present and future," in Proceedings of the 1st Linux Symposium, 2001.
[53] F. Qian, S. Sen, and O. Spatscheck, "Characterizing resource usage for mobile web browsing," in Proceedings of the 12th Annual ACM International Conference on Mobile Systems, Applications, and Services, 2014.
[54] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in Proceedings of the 9th ACM International Conference on Mobile Systems, Applications, and Services, 2011.
[55] J. Ren, L. Gao, H. Wang, and Z. Wang, "Optimise web browsing on heterogeneous mobile platforms: A machine learning based approach," in Proceedings of the 36th IEEE Conference on Computer Communications, 2017.
[56] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and G. Dan, "Enerj:approximate data types for safe and general low-power computation," in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.
[57] D. Shingari, A. Arunkumar, B. Gaudette, S. Vrudhula, and C.-J. Wu, "Dora: optimizing smartphone energy efficiency and web browser performance under interference," in Proceedings of the 16th IEEE International Symposium on Performance Analysis of Systems and Software, 2018.
[58] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.
[59] S. Singh, "Html5 on the rise: No longer ahead of its time," 2015. http://techcrunch.com/2015/10/28/html5-on-the-rise-no-longer-ahead-of-its-time/
[60] G. Sterling, "Morgan Stanley: No, Apps Aren't Winning. The Mobile Browser Is." https://marketingland.com/morgan-stanley-no-apps-arent-winning-the-mobile-browser-is-144303
[61] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with shandian." in Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, 2016.
[62] WikiChip, "A11 Bionic - Apple." https://en.wikichip.org/wiki/apple/ax/a11
[63] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia et al., "Machine learning at facebook: Understanding inference at the edge," in Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture, 2019.
[64] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: Opportunities and limits," in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.
[65] K. Yan, X. Zhang, J. Tan, and X. Fu, "Redefining qos and customizing the power management policy to satisfy individual mobile users," in Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016.
[66] A. Zare, A. Aminlou, M. M. Hannuksela, and M. Gabbouj, "Hevc-compliant tile-based streaming of panoramic video for virtual reality applications," in Proceedings of 24th the ACM on Multimedia Conference, 2016.
[67] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in Proceedings of the 19th ACM International Conference on World Wide Web, 2010.
[68] B. Zhao, W. Hu, Q. Zheng, and G. Cao, "Energy-aware web browsing on smartphones," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 3, pp. 761–774, 2015.
[69] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture, 2015.
[70] ——, "The role of the cpu in energy-efficient mobile web browsing," Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture, vol. 35, no. 1, pp. 26–33, 2015.
[71] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture, 2013.
[72] ——, "Webcore: Architectural support for mobile web browsing," in Proceeding of the 41st ACM/IEEE International Symposium on Computer Architecuture, 2014.
[73] ——, "Greenweb: language extensions for energy-efficient mobile web computing," in Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016.
[74] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in Proceedings of the 48th ACM International Symposium on Microarchitecture, 2015.