Camel: Smart, Adaptive Energy Optimization for Mobile Web Interactions

Jie Ren†, Lu Yuan‡, Petteri Nurmi||, Xiaoming Wang†, Miao Ma†, Ling Gao‡,
Zhanyong Tang‡, Jie Zheng‡, Zheng Wang*
†Shannxi Normal University, China, ||University of Helsinki, Finland,
‡Northwest University, China, *University of Leeds, UK
†renjie@snnu.edu.cn, *z.wang5@leeds.ac.uk

***Abstract*—Web technology underpins many interactive mobile applications. However, energy-efficient mobile web interactions is an outstanding challenge. Given the increasing diversity and complexity of mobile hardware, any practical optimization scheme must work for a wide range of users, mobile platforms and web workloads. This paper presents CAMEL, a novel energy optimization system for mobile web interactions. CAMEL leverages machine learning techniques to develop a smart, adaptive scheme to judiciously trade performance for reduced power consumption. Unlike prior work, CAMEL directly models how a given web content affects the user expectation and uses this to guide energy optimization. It goes further by employing transfer learning and conformal predictions to tune a previously learned model in the end-user environment and improve it over time. We apply CAMEL to Chromium and evaluate it on four distinct mobile systems involving 1,000 testing webpages and 30 users. Compared to four state-of-the-art web-event optimizers, CAMEL delivers 22% more energy savings, but with 49% fewer violations on the quality of user experience, and exhibits orders of magnitudes less overhead when targeting a new computing environment.**

I. INTRODUCTION

Web has become the main approach for accessing information on mobile systems. Indeed, recent studies suggest that 70% of all web traffic comes from mobile devices with the average mobile user in the US spending over three hours per day with web content [1]. When interacting with web content, mobile users want their devices to react fast to interaction events while having a long-lasting battery [2]. Achieving both at once is difficult as web content access often comes with a high energy cost to end-users [3] but existing mechanisms for web access optimization often ignore the effects of energy savings on user experience [4]–[7].

Prior work on energy optimization for mobile web access has predominantly focused on lowering power consumption of the transmission and rendering operations for loading a web page [4], [5], [8]–[10]. Unfortunately, these approaches can only achieve modest savings as they ignore the continuous nature of web interactions. Due to small-form-factor of mobile devices, the webpages often can only be seen through multiple user interactions, such as scrolling and zooming. As we will show in this paper, these operations can consume 2 to 5 times more energy than the initial page loading phase and hence optimizing energy drain of these operations is critical.

Some more recent works try to reduce the energy footprint of web interactions by limiting the processor clock speed [6], [7], [11] or dropping some interaction events [12]. However, these solutions are suboptimal as they achieve energy savings at the cost of user experience. Indeed, the user's sensitivity to response delay differs, with the content type, nature of interactions and interactive speed all affecting user expectations [13]. Another drawback of all existing approaches is that they offer little ability to adapt a decision policy across different computing environments. As mobile hardware, web workloads, and operating system internal mechanisms change over time, it is unlikely that a policy developed today will remain suitable for tomorrow.

We present CAMEL, a novel energy optimization strategy for mobile web interactions that takes into consideration both the need to reduce the energy footprint and to provide good user experience. CAMEL preserves user experience through machine learning models learned *offline* and deployed on the device. These models capture subtle interactions between web content and user perception of delay. This enables CAMEL to make energy-efficient scheduling decisions for any *new* webpage *unseen at design time*. Specifically, CAMEL integrates two types of machine-learning models: a per-user specific predictor that estimates the minimum acceptable response delay for given web content, and a profit estimator that assesses the outcome of different scheduling decisions given the expected user interaction requirements. CAMEL uses these two predictors to quickly find the optimal processing configuration that consumes the least energy but still meeting the interactivity target of the user.

Developing a practical machine learning approach that can generalize across a diverse range of constantly evolving hardware architectures, web workloads and user habits is far from trivial. Prior work has addressed this portability issue through rewriting or retraining [14]. Both solutions, however, are inadequate for mobile web optimization as they either require expert involvement or substantial overhead for training data collection. CAMEL is designed to avoid this pitfall.

To target a diverse set of users and mobile devices, CAMEL employs a novel transfer learning [15] based approach to tune a generic model developed by the application vendor to match the requirements of a new user or hardware platform. Our insight is that one can re-use the knowledge previously obtained from a different platform or user to speed up learning in a new environment considerably. Instead of gathering training samples by profiling the entire dataset, we use statistical methods to determine which part of the dataset is likely to offer useful representative information. By using fewer training instances, we reduce the profiling times and end-user involvement as well as the cost associated with them. We show that despite using many fewer training instances, the resultant performance of CAMEL is comparable to retraining from scratch by profiling the entire dataset.

To adapt to changes in the deployment environment, CAMEL combines statistical and probabilistic assessments to estimate the error bound (or credibility) of a prediction. This provides a rigorous methodology to quantify how much we should trust a model's output, allowing a learning framework to use feedback on uncertain inputs to continuously update a decision model in the end-user environment.

We demonstrate the benefits of CAMEL by integrating it into the rendering architecture of Chromium [16] and evaluating it against four event-based web schedulers [6], [11], [12], [17]. We perform an unprecedentedly large-scale evaluation involving 1,000 testing webpages, 30 users and four distinct mobile devices. Experimental results show that CAMEL consistently outperforms all existing schemes by delivering better energy savings with less frequent violations on the quality-of-experience (QoE). We consider the cases for porting an existing model to different users or hardware. We show that CAMEL provides portable performance but incurs significantly less training overhead over prior strategies.

Contributions. This paper is the first to:

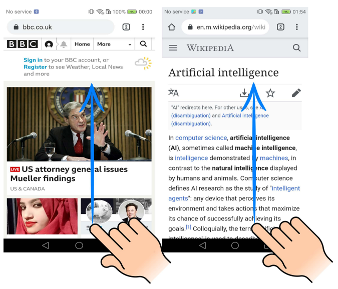
- show how a content-aware QoE optimizing scheme can be developed for web interactions using predictive modeling (Section IV);
- employ transfer learning to address the model portability issue across users and devices (Section V-A);
- exploit statistical assessments to detect and improve ageing models for mobile web browsing (Section V-B).

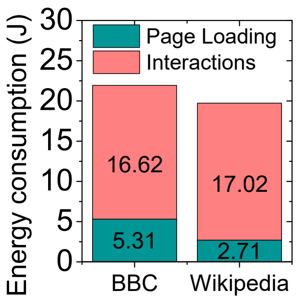
II. Background and Motivation

CAMEL reduces energy usage during web interactions. Existing works largely optimize the initial page loading phase, but as we demonstrate below, interactions have higher energy drain and thus more potential for savings. The few works [6], [11], [17] to address interactions assume a fixed response deadline for web content, but this runs the risk of degrading the overall user experience. By contrast, CAMEL minimizes energy consumption without compromising QoE, by offering "sufficiently good" performance. This is motivated by user experience studies showing that improvements beyond "good enough" are not guaranteed to enhance mobile user experience [2], [18], e.g., the user cannot tell the difference between a lag of 10ms compared to a lag of 100ms [13].

A. Problem Scope

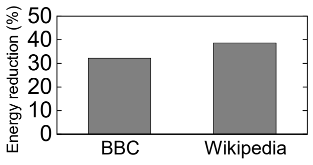
CAMEL targets interactions taking place after web contents have been fetched and the Document Object Model (DOM) tree constructed. We consider three representative browsing gestures: scrolling, pinching (i.e., zoom in and out), and flinging (a quick swipe). We do not consider clicking because it often leads to a new page loading which can be optimized by a page-loading-specific scheme like [8]. Our work targets the widely used big.LITTLE [19] mobile architecture. As a case study, we apply CAMEL to Chromium, the open-source project behind Google Chrome and many other browsers like Opera and Microsoft Edge for ARM. Note that as CAMEL targets response to interaction events within the web rendering engine, it is not restricted to browsers but equally applicable to webview-based apps like social media and newsreaders.


(a) Scrolling up webpages

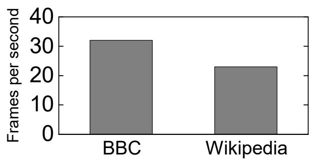


(b) Energy consumption

Figure 1: Motivation webpages (a) and the breakdown of energy consumption during page loading and interactions (b).



(a) Energy reduction



(b) Frames per second (FPS)

Figure 2: Energy reduction (a) and FPS (b) on BBC and Wikipedia pages when using the optimal policy over the `interactive` governor.

Table I: Optimal configurations for motivation examples.

	Event-res. freq.	big CPU (GHz)	little CPU (GHz)	GPU (MHz)
BBC News	1 / 6	1.28	0.672	250
Wikipedia	1 / 15	1.05	0.49	250
render process placement			✓	

B. Motivating Examples

Consider the scenario depicted in Figure 1 (a) where a user is scrolling up when reading two webpages from BBC News and Wikipedia on a recent XiaoMi 9 smartphone (detailed in Section VI). Here, we use RERAN [20], a record and replay tool, to replay user interactions.

1) Energy consumption: interactions vs page loading. Figure 1(b) compares the energy consumed in response to scrolling against that spent during the loading phase in a WiFi environment. The measurement excludes energy consumption during CPU and GPU idle time. To minimize the impact of screen use and background workloads, we set the screen to the lowest brightness and close all background workloads. As can be seen from the diagram, the energy spent during the interaction phase is 2-5 times higher than that used in the initial loading phase. This finding is in line with prior studies [11], [21], suggesting that prior approaches that only focus on the loading phase would miss a massive optimization opportunity.

2) Room for improvement. In the second experiment, we wish to understand how much room is available for trading performance for reduced energy consumption. We consider two established techniques: (1) setting the CPU/GPU frequency and running the render process on the big or little CPU cluster, and (2) dropping some of the interaction events (i.e., approximate computing). To quantify the user expectation, we use frames per second (FPS), because it is shown to strongly correlates to the user's perceived responsiveness for web browsing [6], [11]. For most of the participants in our

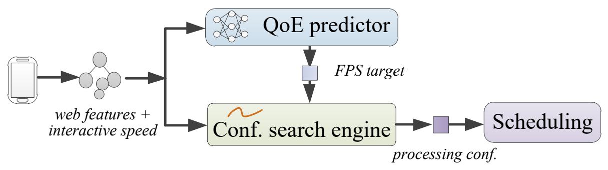


Figure 3: Overview of the scheduling framework of CAMEL.

user study (see Section IV-A2), the minimum acceptable FPS for the BBC and Wikipedia pages is 32 and 23 respectively. The disparity in the tolerable FPS is due to the content of the two pages. The BBC page is dominated by images while the Wikipedia one is dominated by dense texts, and human eyes are less sensitive to the change of text-dominated content [22].

Figure 2 gives the energy reduction achieved by the optimal policy over the Android default `interactive` frequency governor and the resultant FPS. To find the optimal policy, we automatically replay the scrolling events and exhaustively profile all possible options. Table I lists the best processing configurations for the testing webpages. The best policy for the BBC page is to respond to one of every six input events[1] and run the render process on the little CPU with an appropriate clock speed for CPUs and the GPU. This configuration reduces energy consumption by 32.2%. For the Wikipedia page, the best policy gives an energy saving of 38.6%. However, applying the best policy of the Wikipedia webpage to the BBC one will give an FPS of 26 (6 FPS below the target of 32) and a modest energy saving of 2.6% over the actual optimal policy. Therefore, simply using one optimal policy found for one webpage to another is likely to either miss optimization opportunities or compromise QoE.

3) Insights. The two examples show the enormous potential of energy optimization for mobile web interactions. However, finding the right processing setting is difficult as it depends on the web content, individual user expectation, and hardware. In the next section, we will describe how CAMEL addresses this challenge by directly modeling the impact of the web content and interactive speed on user acceptable delay through predictive modeling.

III. OVERVIEW OF CAMEL

Figure 3 depicts the scheduling framework of CAMEL. It consists of two innovative components: (a) a QoE predictor to estimate the minimum acceptable FPS target for a given user, interactive speed and web content, and (b) a configuration search engine to find a *processing configuration* (i.e., *an event-response frequency and a processor setting*) that meets the minimum FPS constraint with minimal energy usage.

A. QoE Predictor

Our QoE predictor takes as input features of the web page and the incoming interactive speed. It then predicts the minimum acceptable FPS. A baseline predictor for each targeting event was first trained "*at the factory*" through a user study. The baseline predictor then continuously improves itself for each target user after deployment.

[1]Depending on the speed and duration, a gesture often generates multiple events. For example, a flinging action can trigger over 70 scrolling events.

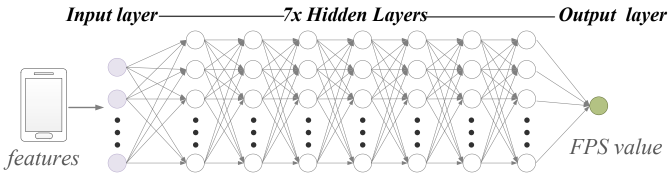


Figure 4: Our neural network based predictor.

B. Configuration Search Engine

Given a content-specific QoE constraint expressed as an FPS target, the configuration search engine finds a processing configuration to use. This is achieved by using an FPS predictor (or profit estimator) to estimate the FPS as a function of a processing configuration and web features. By varying the processing configuration given to the predictor, the search engine can then exam the expected FPS and choose the best-performing configuration before applying it. The chosen processing configuration is passed to the render-related processes and a runtime scheduler to adjust the event-response frequency and processor settings. Like the QoE predictor, we learn one FPS predictor for each event type, three in total.

C. Adaptive Learning

CAMEL is designed to be a practical scheme that is portable across users and mobile devices. There are two critical challenges related to this design goal. Firstly, how to reduce the end-user involvement in capturing a user's QoE requirement. Secondly, how to detect and improve an ageing decision model in the deployment environment.

To reduce end-user involvement, CAMEL employs transfer learning (Section V-A) to quickly re-target an existing model for a *new* user or platform. Rather than retraining on the entire training dataset, transfer learning uses only a dozen of webpages. This not only significantly reduces the profiling overhead but also allows performing learning on the user's device to mitigate the privacy concern for doing that on a remote server [12]. To detect and improve ageing models, CAMEL uses conformal predictions (Section V-B) to assess the credibility of each prediction. It then uses user feedback or automated runtime measurements on incorrectly predicted inputs to improve a deployed model over time. This continuous learning strategy minimizes user intervention by only asking for feedback when things have gone wrong.

IV. PREDICTIVE MODELING

The QoE and FPS predictors employed by CAMEL are artificial neural networks (ANNs). We choose the ANN because it gives better and more robust performance over alternatives (Section VII-E), and also allows the use of transfer learning to mitigate the training overhead in the deployment environment (Section VII-C). We describe our predictive modeling based framework by following the classical 3-step process for supervised learning: (1) problem modeling and training data generation (2) train a predictor (3) use the predictor.

A. Problem Modeling and Training Data Generation

1) Model structure. Figure 4 depicts our neural network - a fully connected, feed-forward ANN with 7 hidden layers and

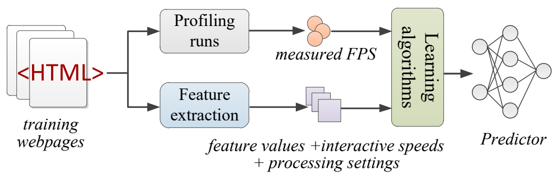


Figure 5: Overview of our training process for FPS predictions.

260 nodes per hidden layer. The number of nodes of the input layer is determined by the dimensionality of the model input (Section IV-A3). This structure is *automatically determined* by applying the AutoKeras [23] AutoML tool on the training dataset. In Section VII-E, we evaluate the impact of network structures on performance.

2) Training data generation. We apply cross-validation to train and test our models (see also Section VI-C). Training data are generated by profiling a set of training webpages.

FPS training data. Figure 5 depicts the process for learning a baseline FPS predictor on 800 training webpages. To generate training data, we use RERAN to automatically generate a gesture at different speeds (measured by the number of pixels per second) on each training webpage. For each interactive speed, we vary the processing configurations and record the achieved FPS. Specifically, we exhaustively execute the computation-intensive render and paint processes under each CPU/GPU setting. We also evaluate all candidate event-response frequencies for a processor setting. In total, we train an FPS predictor on over 1 million *automatically generated* training samples (800 webpages $\times$ 10 interactive speeds $\times$ $\sim$ 16 processor settings $\times$ 8 event-response frequencies). The processor settings and event-response frequencies are configurations on the optimal frontier of performance and energy trade-offs, which are determined by profiling all possible settings on 20 randomly chosen training webpages. Note that the trained model can be applied to arbitrary interactive speeds and processor settings by taking these as the model inputs. Finally, for each webpage, we collect the web features as we will described later in this section. We stress that this process is *fully automated* and does not require user involvement.

QoE training data. Our QoE training data are gathered through a user study. In practice, this can be done through a crowdsourcing platform like Amazon Mechanical Turk. Our user study involved 30 paid users (15 females) who were studying at our institution. To minimize user involvement, we apply the *k-means* clustering algorithm [24] to choose 100 representative webpages from our training dataset. We ask each user to watch the screen update of each training webpage on a XiaoMi 9 smartphone under various FPS speeds. We also vary the incoming event by considering 5 commonly interactive speeds per gesture [12] To help our participants to correlate the generated events to finger movements, we invite them to interact with the device and show the resultant FPS of their finger movements. For each training instance, we ask a user to select the lowest acceptable screen update rate. We then record the corresponding minimum acceptable FPS on a per-webpage, per-speed, per-gesture and per-user basis. On average, it took a participant 2.5 hours to complete the study. Later, we extend this user study to all 1,000 webpages used for QoE evaluation using cross-validation.

Table II: Raw web features used in the work

DOM Tree	#DOM nodes #each HTML tag	depth of tree #each HTML attr.
Style Rules	#rules #each selector pattern	#each property
Other	GPU memory footprint for viewports	

3) Feature extraction. One of the key aspects in building a good predictor is finding the right features to characterize the input workload. In this work, we started from 6,913 raw web features extracted from the DOM tree. Table II summarizes our raw features. The features were chosen based on previous work of mobile web optimization [8] and our intuitions.

The QoE model takes as input the web features of current and future viewports and the user interactive speed. The FPS model takes as input the web features, the interactive speed, the processing setting (i.e., *event-response frequency and processor setting*), and the CPU cluster where the render process is running on (for modeling the penalty for cross-processor task migration).

Feature reduction. To learn effectively over a small training dataset, we apply the correlation coefficient and principal component analysis [24] to reduce the dimensionality of raw web features from 6,913 to 127. Both techniques are shown to be useful in prior work for feature reduction [8], [25].

Feature normalization. In the final step, we scale each of the extracted feature values to a common range (between 0 and 1) to prevent the range of any single feature being a factor in its importance. We record the minimum and maximum values of each feature in the training dataset, in order to scale the feature values of an unseen webpage. We also clip a feature value to make sure it is within the expected range during deployment.

4) Training overhead. The time for training the baseline predictors is dominated by generating the training data. In this work, it takes less than a week to collect all the training data for a mobile platform. In comparison processing the raw data, and building the models took a negligible amount of time, less than an hour for learning all individual models on a PC. We stress that training of the baseline predictors is a one-off cost.

B. Training a Baseline Predictor

The collected web feature values and speed together with the desired FPS values are passed to a supervised learning algorithm to learn an ANN for each event. For FPS predictions, we also use additional model inputs as stated in Section IV-A3. Our models are trained using back-propagation with stochastic gradient descent (SGD) guided by the widely used Adam optimizer [26] and L2 regularization, which is a standard-setting for training ANNs. For training examples $y_1 \ldots y_n$, the optimizer finds model parameters Θ to minimize the output of a mean-squared-logarithmic loss (MSLE) function ℓ:

$$\Theta = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, \Theta\right)$$

We choose MSLE because it penalizes underestimates more than overestimates, which reduces the chance of QoE violations due to an underestimated FPS target.

C. Using the Models

The trained predictors can be applied to *new, unseen* webpages. We implemented our models using Keras [27] and Scikit-learn [28]. Our optimization can be turned on by the user or enabled during Android's "Battery Saver" mode. When a supported event is detected, we will extract web features of the current and the future viewports from the DOM tree – the future viewport is calculated based on the interactive speed. We calculate the average interactive speed using a sampling window of 50 ms or the interactive session – which is shorter. We then use the QoE and FPS predictors to choose the optimal processing configuration as described in Section III-B. To minimize the runtime overhead, our framework runs on the least loaded CPU core. The overhead of feature extraction, predictions, searching, and runtime scheduling is small – less than 5 ms, which is included in all experimental results.

V. Adaptive Learning

We propose two new ways to improve the adaptiveness and practicability of a machine-learning-based web optimizer.

A. Adapt to A New Environment

1) The problem. QoE is user-specific and the resultant FPS depends on the underlying hardware. Therefore, using a generic model across different users and hardware platforms is ineffective. To tune a model to match a specific user or mobile device, CAMEL employs transfer learning [15] to quickly port a baseline predictor to the target computing environment.

2) The idea. Prior work in other domains has shown that ANN models trained on similar inputs for different tasks often share useful commonalities [29]. Our work leverages this insight to speed up the process for tuning a model for a new user or mobile hardware. This is because the first few layers (i.e., those close to the input layer) of our ANN are likely to focus on abstracting web features and largely independent of the model output. Since we use the same network structure, transfer learning is achieved by copying the weights of a baseline model to initialize the new network. Then, we train the model as usual but using profiling information (as described in Section IV-A) collected from fewer training webpages.

3) Determining training samples. A key question for applying transfer learning in our context is how many training examples do we need. Under-provisioning of training data will lead to low accuracy, while over-provisioning will incur significant profiling overhead especially when that requires end-user involvement. To determine the right number of training examples, we group our training webpages using the *k-means* clustering algorithm. We then choose two webpages from each cluster: one is mostly close to its cluster centroid on the feature space, the other has the biggest Frobenius norm value [30] with respect to other centroid points. In practice, the chosen webpages can be shipped as part of the browser bundle, where profiling can be performed when the device is charging after the first installation.

To determine the right number of clusters (i.e., K), we use the *Bayesian Information Criterion* (BIC) score [31]. The BIC measures if a selected K is the best fit for grouping data samples within a dataset. The larger the score is, the higher the chance that we find a good clustering number for the dataset. The BIC score is calculated as [32]:

$$BIC_j = \hat{l}_j - \frac{p_j}{2} \cdot logR$$

where $\hat{l}_j$ is the likelihood of the data when K equals to j, R is the number of training samples, and the free parameter p_j is the sum of $K-1$ class probabilities – calculated as: $p_j = (K-1) + dK + 1$ for a d-dimension feature vector plus 1 variance estimate. $\hat{l}_j$ is computed as:

$$\hat{l}_j = \sum_{n=1}^{k} -\frac{R_n}{2} log(2\pi) - \frac{R_n \cdot d}{2} log(\hat{\sigma}^2) - \frac{R_n - K}{2} + R_n log(R_n/R)$$

where R_n is the number of points in a cluster and $\hat{\sigma}^2$ is the distance variance between each point to its cluster centroid.

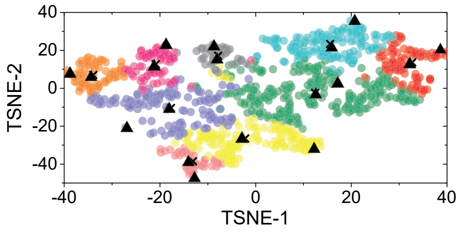


Figure 6: Using clustering to choose training examples for transfer learning. A cluster centroid is marked by a cross, while the two chosen webpages of a cluster are marked as triangles.

4) Illustrative example. Figure 6 illustrates how one of our training dataset of 800 webpages can be grouped into 9 clusters determined using the BIC score. To aid the clarity, we apply t-SNE [33] to project the data onto a two-dimensional space. Directly using an FPS model trained for Pixel 2 to XiaoMi 9 gives an error rate of 37.5%. By using profiling information collected from 18 carefully chosen webpages on the target device to update the predictor, the error rate decreases to 6.7%. Such performance is not far from the error rate of 4.6% when training the model from scratch by profiling the entire 800 training webpages, but we cut the training time from two days to less than two minutes on the end user's phone.

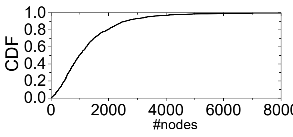
B. Continuous Learning at the Deployment Environment

1) The problem. The key for continuously improving a model after deployment is knowing when the model is wrong so that we can use the ground-truth to improve it. Judging if an FPS prediction is inaccurate is straightforward because the ground-truth can be automatically measured. Checking if a QoE target meets the user expectation is harder because we cannot ask a user to provide feedback every time.

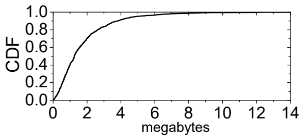
2) The solution. To estimate if a QoE target prediction is wrong, we leverage the conformal prediction (CP) [34], [35]. The CP is a statistical assessment method for quantifying how much we could trust a model's prediction. This is done by learning a *nonconformity function* from the model's training data. This function estimates the "strangeness" of a mapping from input features, x, to a prediction output, y, by looking at the input and the probability distribution of the model prediction. In our case, the function estimates the error bound of a QoE prediction. If the error bound is greater than a

Table III: Evaluation platforms

Device	CPU	GPU	RAM (GB)	Screen (inches)	OS
XiaoMi 9	Snapdragon 855 @ 2.84 GHz	Adreno 640	8	6.39	MIUI 10 (Android 9)
Google Pixel 2	Snapdragon 835 @ 2.35 GHz	Adreno 540	4	5.0	Android 9
Huawei P9	Kirin 955 @ 2.5 GHz	Mali T880	3	5.2	Android 8
Odroid Xu3	Exynos 5422 @ 2 GHz	Mali T628	2	4	Ubuntu 16.04



(a) #DOM nodes



(b) Webpage size

Figure 7: The CDF of #DOM nodes (a), webpage size (b).

configurable threshold (20% in this work), we then consider the model gives an incorrect prediction.

CAMEL uses the inductive CP as it works with any regression model [36]. For a prediction, y, of input x, function f calculates the nonconformity score as:

$$f(x,y) = \frac{|y - h(x)|}{g(x) + \beta}$$

where h is a regression-based QoE or FPS model, g estimates the difficulty of predicting y and β is a sensitive parameter that determines the impact of normalization. Note that g and β are automatically determined from training data.

3) Continuous learning. For a QoE prediction that is considered to be inaccurate, CAMEL takes the high-end value of the CP-estimated error bound to minimize QoE violations. It then finds out the actual QoE target by seeking user feedback. This is done by automatically replaying the screen update under different FPS settings, from high to low. For each setting, CAMEL asks the user to rate the screen update for being "acceptable" or not. It stops playing the screen update when the user indicates an FPS setting is unacceptable. To update a QoE or FPS model, CAMEL adds profiling information of the uncertain inputs to the transfer learning dataset. When the device is charging, CAMEL runs the learning algorithm to update the predictors and CP models.

VI. EVALUATION SETUP

A. Platforms and Workloads

Evaluation Platforms. To implement CAMEL, we modified Chromium (ver. 74)[2] and compiled it under the "release" build. Our evaluation platforms, detailed in Table III, include different hardware specs, representing low, medium and high-end mobile systems. We specifically include Odroid Xu3, because all, except one [12], of our competitive schemes have been tuned and evaluated on this platform.

Web Workloads. We use the landing page of the top 1,000 hottest websites (as of May, 2019) ranked by `alexa.com` based on the global web traffic analysis. Figure 7 shows the CDF of the number of DOM nodes and web content sizes. The webpage sizes range from small (4 DOM nodes and 10 KB) to large (over 7000 DOM nodes and 14 MB), indicating that our test data cover a diverse set of web contents.

[2]Code can be downloaded from *[https://bit.ly/2srZbs9]*.

B. Competitive Approaches

We compare CAMEL to the following state-of-the-arts:

- `EBS`: A regression-based method for adjusting the processor frequency to meet a fixed response deadline [6];
- `Phase-aware`: An event-phase-based power management strategy for mobile web browsing [17];
- `ML-governor`: A machine-learning-based CPU frequency governor for interactive web browsing [11];
- `eBrowser`: This strategy puts the browser process into sleep to drop some of the input user events [12].

All the above schemes require learning on the entire training dataset for each hardware architecture. Moreover, all, except eBrowser, assume a fixed deadline for an event type.

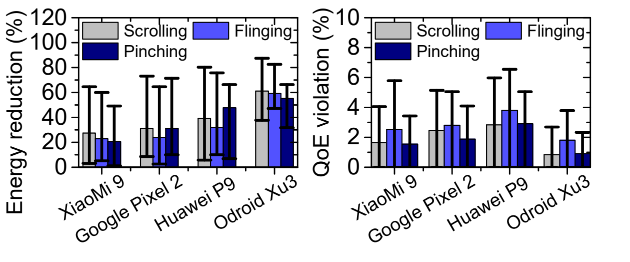
C. Evaluation Methodology

Model evaluation. Like [11], we use five-fold cross-validation to train all machine learning models (including our competitors). Specifically, we randomly partition our 1,000 websites into 5 sets where each set contains webpages from 200 sites. We keep one set as the validation data for testing our model, and the remaining 4 sets as training data to learn a model. We repeat this process five times (folds) to make sure that each of the 5 sets used exactly once as the validation data. To minimize user involvement, we use a subset of webpages from the training dataset to build the QoE model as described in Section IV-A2. This is a standard methodology for evaluating the generalization ability of a learned model.

Metrics. We consider two metrics: energy saving and QoE violation. Energy saving is normalized to the energy consumed by the `interactive` scheduler, an Android default CPU governor for interactive applications. QoE violation is calculated as δ/FPS_{min}, where δ is the number of FPS falls below the minimum acceptable FPS, FPS_{min} [11]. We do not use `powersave` as a baseline as it gives long processing times and violates QoE for all our test cases.

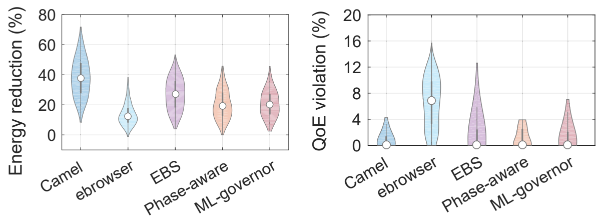
Measurements. For energy measuring, we use a Monsoon power meter [37] (except for Odroid Xu3 because it already has onboard power sensors for energy measurement) to measure the power consumption of the *entire* system including the display with a 50% brightness (a typical indoor setting of Android). For the FPS, we use a script to count the number of invocations of the `SurfaceView` object of Chromium.

Reporting. When reporting performance, we use the *geometric mean*, which is widely seen as a more reliable performance metric over the arithmetic mean [38]. Unless state otherwise, we report the geometric mean across 3.6 million automatically-generated test cases of 1,000 webpages, 30 users, 4 devices, 3 gestures and 10 speeds per gestures, using cross-validation. Moreover, events are automatically generated, starting from the initial viewport of a webpage. To have statistically sound data, we run each approach on a test case repeatedly until the confidence-bound under a 95% confidence interval is smaller than 2%. Finally, all webpages are loaded from the device's



(a) Energy reduction (b) QoE violation

Figure 8: The energy reduction (a) and QoE violations (b) achieved by our approach over `interactive`.



(a) Energy reduction (b) QoE violation

Figure 9: Compare with the state-of-the-arts. CAMEL consistently outperforms all alternatives.

internal storage to preclude network variances, and we disable the browser cache to ensure consistent results across runs.

VII. EXPERIMENTAL RESULTS

A. *Content-aware QoE Optimization*

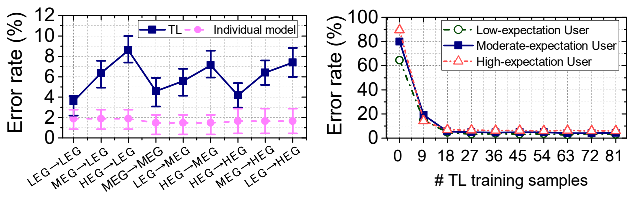
To evaluate the benefit of content-aware QoE optimizations, in this experiment we train our predictors on the entire training dataset, but we will evaluate transfer learning in Section VII-C. The results are given in Figure 8, where the min-max bars show the variances across our evaluation scenarios.

Figure 8a shows that CAMEL reduces energy consumption by at least 23.6% (up to 58.5%), and Figure 8b confirms that such large energy reduction does not necessarily come at the cost of poor user experience. CAMEL only leads to 1% to 4% of QoE violations on less than 5% of the testing webpages with 2 to 3 lower than expected FPS values. On testing webpages where no QoE violation occurred, CAMEL delivers 92.4% of the available energy savings given by a *theoretically perfect* predictor (found by exhaustively profiling) that always chooses the optimal processing configuration. Furthermore, if we take a conservative approach by adding 10% to the predicted FPS QoE target, CAMEL can then eliminate all the QoE violations, but still gives an average energy reduction of 21.3% (12.1% to 37.4%). This results show that CAMEL is highly effective in trading performance for energy savings.

B. *Compare to Competitive Approaches*

Figure 9 compares CAMEL with alternative schemes. The white dot in the plots denotes the median value and the thick black line represents 50% of the data. For fair comparison, all schemes are built from the same training dataset.

All approaches improve over the `interactive` baseline. By modeling the impact of web content on QoE and using this to configure the heterogenous hardware, CAMEL gives the highest overall energy saving and the lowest QoE violation ratio. Specifically, CAMEL reduces the energy consumption by at least 14.6% (up to 29%), but with at least 25.1% (up to 88.3%) lower QoE violations compared to prior methods.



(a) TL within and across groups (b) Perf. w.r.t. #webpages

Figure 11: Applying transfer learning for QoE predictions.

C. *Evaluation of transfer learning*

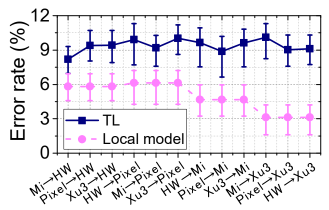
We now evaluate our strategy for applying transfer learning (TL) to tune baseline predictors for a new environment. On average, TL delivers 27.4% (up to 53.1%) of energy savings with less than 6% of QoE violations. This performance is comparable to the one reported in Section VII-A when the QoE and FPS predictors are trained from scratch every time.

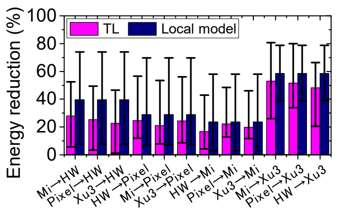
1) Tuning FPS predictors. Figure 10 shows the results for using TL to port a baseline FPS predictor for a new platform. Although we only use 2.3% of the training examples (i.e., 18 webpages - see Section V-A3), performance of the TL-learnt model is compared to training a completely new model using 800 webpages. We see only a marginal increase of 4.46% in the error rate. As subgraphs b and c in Figure 10 show, on average, TL gives 29.7% of energy reduction with 4.9% of QoE violations for porting an FPS predictor to a new platform.

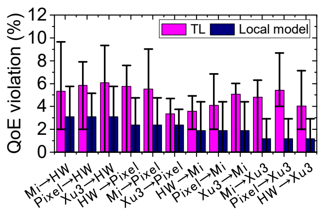
Figure 10d shows how the error rate changes as we increase the number of training webpages when using TL to port an FPS model built for XiaoMi 9 to Huawei P9. Using more webpages does improve prediction accuracy. However, the performance reaches a plateau when using 18 webpages, and a further increase in the number of training webpages does not justify the increased profiling overhead.

2) Tuning QoE predictors. We divide the 30 participants of our user study into 3 groups based on their minimum acceptable FPS. The low-expectation group has 10 users with an averaged FPS target of under 35; the moderate-expectation group has 14 users with an averaged FPS target of between 35 and 49; and the high-expectation group has 6 users with an averaged FPS target of over 49.

Figure 11a reports the performance for applying TL (with cross-validation) to port a QoE predictor to another user from the same or a different group. As expected, TL within the same user group gives the lowest error rate of between 3.1% (1.1 FPS) and 4.58% (2.08 FPS). We see a slight increase in the error rate when applying TL across user groups, but the average error rate is 6.9% (2.94 FPS). In practice, we can further improve the performance by choosing a pre-trained model that is as close as possible to the target user based on observations seen from the first few webpages, e.g., using a


(a) Error rate


(b) Energy reduction


(c) QoE violation

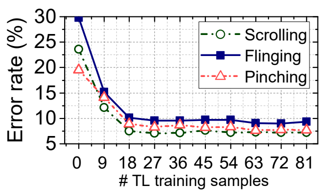

(d) From Mi9 to HW P9

Figure 10: Transfer learning across platforms. We can use profiling information collected from 18 webpages to update a model.

recommendation system [39], [40]. We leave this as our future work. Figure 11b shows the error rate when applying TL to a QoE model from a different group to the LEG group. Like the previous experiment, we see the accuracy improvement reaches a plateau when using 18 webpages.

D. *Evaluation of Continuous Learning*

To mimic the impact of changing web workloads on a deployed QoE or FPS predictor, we train an *initial* predictor on 50% of the training samples and test the trained predictor on the remaining webpages using cross-validation. To isolate the impact of TL, the initial models in this evaluation are learned using data collected from the target environment.

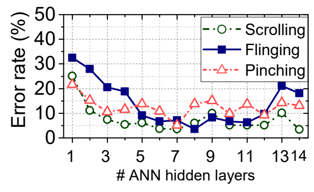
Detect ageing models. Our first experiment aims to evaluate CAMEL's ability in using CP to detect an ageing QoE predictor due to workload changes. We do not apply CP to the FPS predictor because the ground-truth can be directly measured. We are interested in knowing how often our CP function (see Section V-B) successfully detects when a predicted QoE target has an error of more than 5%. Our CP scheme successfully catches 96.4% of the inputs where the QoE predictor gives a wrong prediction under our criterion. Our scheme also has a low false positive (i.e., when the CP model thinks the QoE predictor is wrong but it is not) rate of 5%.

Model update. We can use user feedback (for QoE predictions) or automated profiling information (for FPS predictions) on the first few mispredicted webpages flagged by CAMEL to update an existing model. We found that CAMEL updated using five mispredicted webpages delivers on average 98% (for QoE predictions), and 97% (for FPS predictions) of the performance given by a model trained using the entire dataset on the target platform. This translates into an improvement of over 23.4% for the initial predictor in this experimental setting. Because profiling only needs to be performed on incorrectly predicted inputs, the model retraining process is fast, taking less than 2 minutes on a XiaoMi 9 phone; in comparison, profiling on the entire training dataset would take hours.

In practice, one would first use TL to tune the baseline predictors during the first installation. Then, the CF scheme can be used to update the installed models. This experiment shows that CAMEL is highly effective in detecting and updating ageing models without incurring significant overhead.

E. *Model Analysis*

1) Impact of neural layers and training samples. Figure 12a gives the error rate when an ANN-based FPS predictor is constructed with different numbers of hidden layers. To isolate the impact of TL, we first train the model using 800 webpages and then test the trained model on another 200 webpages. Using 7, 14 and 8 hidden layers give the best performance for scrolling, flinging and pinching respectively. We choose to use a unified model structure with 7 hidden layers as it requires fewer training examples and the performance is not far from the optimal settings. Looking at Figure 12b, we see a steady decline in error rates when using more examples to train the baseline predictors. This is not surprising, as the performance of a predictive model generally improves as the number of training samples increases. Since a baseline model only needs to be trained once, this is a one-off cost.


(a) # Training hidden layers

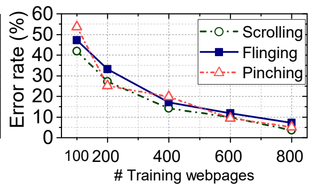
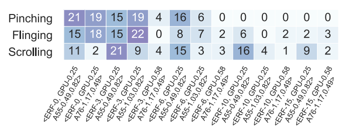
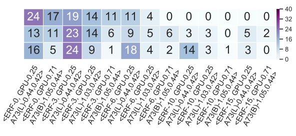
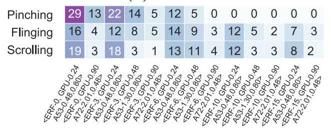

(b) Perf. w.r.t. #webpages

Figure 12: Impact of the number of hidden neural layers (a) and training webpages for our ANN-based FPS predictors (b).


(a) Xiaomi 9


(b) Google Pixel 2


(c) Huawei P9

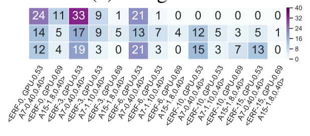

(d) Odroid Xu3

Figure 13: Distributions of optimal processor settings.

2) Processor configuration distributions. Figure 13 shows the distribution of the most optimal processor settings. Here, we use the notation < ERF - event response frequency, GPU-freq, rendering CPU core - rendering CPU core freq, other CPU core freq> to denote a processing configuration. For example, <ERF-10, GPU-0.48, A53-1.3, 0.8> means that we response to 1 out of every 10 input events of the same type, the painting process running on the GPU at 480MHz, and the render process running on the little A53 core at 1.3 GHz while the big core operates at 800MHz. Although some of the configurations are being optimal more frequently than

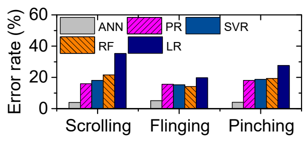


Figure 14: Comparing our ANN-based predictor with other modeling techniques.

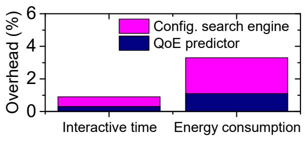


Figure 15: Breakdown of runtime overhead.

others, the distribution varies across event types and hardware platforms. This diagram reinforces the need for an adaptive scheme. CAMEL is designed to offer such a capability.

3) Alternative modeling techniques. Figure 14 compares our ANN-based FPS predictor against four alternative regression methods used in prior relevant works: Linear Regression (LR), Polynomial Regression (PR), Support Vector Regression (SVR), and Random Forest (RF). All the alternative techniques were trained and evaluated by using the same method and training data as our models. Our approach achieves the lowest error rate and enables us to employ transfer learning.

4) Overhead breakdown. Figure 15 gives a breakdown of the runtime overhead of CAMEL (which was already included in our experimental results). CAMEL introduces little overhead to the end to end turnaround time and energy consumption, less than 1% and 4% respectively.

F. Discussions and Future Work

Multi-tasking environment. CAMEL can be extended to a multi-tasking environment for optimizing the front-running application. On mobile systems, background workloads are typically put into a sleeping or closed status, and thus not require a quick response at the background. CAMEL can also be integrated with an interference-aware scheduler like [9] to minimize the impact on concurrently running workloads.

Display optimization. Our experimental results already include energy consumption of the screen, but we do not optimize the display setting. Since the display setting does not affect the processing latency, CAMEL can be easily integrated with a display optimization scheme like [41] and [42].

Dynamic content. Our work does not consider network latency as most of the web content would already be downloaded before a user interaction commences. However, it is possible that user interaction will trigger new network activities. Methods on latency-aware optimizations for page loading [8] or dynamic content [43] are thus complementary to CAMEL.

Apply to other applications. CAMEL can be directly applied to WebView-based applications without modification to the application code. From Android 4.4, WebView is based on the Chromium rendering architecture on which CAMEL is tested.

VIII. Related Work

Our work is broadly related to the literature in four areas:

Mobile web workload optimization. Prior work has focused on the initial page loading phase through e.g., dynamic frequency scaling [4], [8], [9], accumulating traffics [5], [10] and parallel downloading [44]. CAMEL targets the later user interaction phase. It is closely related to event-based power management for mobile web browsing [6], [7], [11]. However, these previous methods have three drawbacks: (1) by assuming a fixed response deadline, (2) have intensive overhead for targeting new hardware and user, and (3) cannot examine whether a decision model still fits. eBrowser [12] uses image entropy to characterize the web content, but it requires all web contents to be rendered ahead of time, introducing significant start up delays and could waste computation cycles. CAMEL is designed to address these limits, offering a better and practical way to target a wider range of computing environments.

QoE modeling. Prior research models user experience through usability studies [45], contextual inquiries [46] or data logging [2], by considering generic metrics like power consumption, response time and network latency. Unlike these works, CAMEL is a content-aware QoE estimation scheme by directly modeling the impact of web workloads on QoE.

Energy optimization. Other relevant works include optimizations for the display [47] and radio [48], dynamic content caching [49] or prefetching [50], optimizations for JavaScript [43], and multi-event scheduling [25]. As pointed out in [50], mobile web browsing requires novel techniques across the computing stack; CAMEL thus benefits from techniques from different computing layers.

Machine learning for systems optimizations. Machine learning has been used to model power consumption [51], task scheduling [14], [52] of mobile systems and program tuning in general [53]–[72]. Our work tackles an outstanding problem of porting a model to a new computing environment. Transfer learning was recently used for wireless sensing [73] through randomly chosen samples. CAMEL improves [73] by carefully choosing representative tracing examples for transfer learning. Conformal prediction was used for malware classification [74], but not the regression problem addressed by CAMEL. We note that the novelty of CAMEL is a new way of combining statistical learning and techniques, rather than improving the learning algorithm itself.

IX. Conclusions

This paper has presented CAMEL, a novel energy optimization scheme for interactive mobile web browsing. Unlike prior work, CAMEL models how the web content and interactive speed affects the QoE. To develop a practical solution, CAMEL employs transfer learning and conformal predictions to automatically adapt an existing policy to the changes of users, hardware platforms or web workloads. We apply CAMEL to Chromium and evaluate it on four mobile systems across 1,000 webpages and 30 users. Experimental results show that CAMEL consistently outperforms existing web-optimizers, and has less overhead when targeting a new user or device.

Acknowledgements

This work was supported in part by the NSF China under grant agreements 61902229, 61872294, 61877038, 61602290 and 61877037; and the Natural Science Basic Research Program of Shaanxi Province under Grant No. 2019JQ-271. For any correspondence, please contact Zheng Wang (E-mail: z.wang5@leeds.ac.uk).

REFERENCES

[1] B. McLeod, "75+ mobile marketing statistics for 2019 and beyond," 2018. [Online]. Available: https://www.bluecorona.com/blog/mobile-marketing-statistics

[2] A. Zuniga *et al.*, "Tortoise or hare? quantifying the effects of performance on mobile app retention," in *WWW*, 2019.

[3] N. Thiagarajan *et al.*, "Who killed my battery?: analyzing mobile browser energy consumption," in *WWW*, 2012.

[4] J. Ren *et al.*, "Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach," in *INFOCOM*, 2017.

[5] D. Li *et al.*, "Automated energy optimization of http requests for mobile applications," in *ICSE*, 2016.

[6] Y. Zhu *et al.*, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *HPCA*, 2015.

[7] N. Peters *et al.*, "Phase-aware web browser power management on hmp platforms," in *ICS*, 2018.

[8] J. Ren *et al.*, "Proteus: Network-aware web browsing on heterogeneous mobile systems," in *CoNEXT*, 2018.

[9] D. Shingari *et al.*, "Dora: Optimizing smartphone energy efficiency and web browser performance under interference," in *ISPASS*, 2018.

[10] W. Hu and G. Cao, "Energy optimization through traffic aggregation in wireless networks," in *INFOCOM*, 2014.

[11] L. Yuan *et al.*, "Using machine learning to optimize web interactions on heterogeneous mobile systems," *IEEE Access*, 2019.

[12] F. Xu *et al.*, "ebrowser: Making human-mobile web interactions energy efficient with event rate learning," in *ICDCS*, 2018.

[13] V. Seeker *et al.*, "Measuring qoe of interactive workloads and characterising frequency governors on mobile devices," in *IISWC*, 2014.

[14] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proc. IEEE*, 2018.

[15] M. Long *et al.*, "Deep transfer learning with joint adaptation networks," in *ICML*, 2017.

[16] T. C. Projects, "Chromium." [Online]. Available: https://www.chromium.org/Home

[17] N. Peters *et al.*, "Phase-aware web browser power management on hmp platforms," in *ICS*, 2018.

[18] Y. Choi *et al.*, "Graphics-aware power governing for mobile devices," in *MobiSys*, 2019.

[19] ARM, "White paper: big.little technology: The future of mobile," 2013.

[20] L. a. Gomez, "Reran: Timing-and touch-sensitive record and replay for android," in *ICSE*, 2013.

[21] A. Roudaki *et al.*, "A classification of web browsing on mobile devices," *Journal of Visual Languages & Computing*, 2015.

[22] P. G. Barten, *Contrast sensitivity of the human eye and its effects on image quality*, 1999, vol. 21.

[23] H. Jin *et al.* (2018) Auto-keras: An efficient neural architecture search system.

[24] C. M. Bishop, *Pattern Recognition and Machine Learning*, 2006.

[25] Y. Feng and Y. Zhu, "Pes: Proactive event scheduling for responsive and energy-efficient mobile web computing," in *ISCA*, 2019.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv*, 2014.

[27] "Keras: The Python Deep Learning library," https://keras.io/.

[28] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, 2011.

[29] T.-K. Hu *et al.*, "Learning adaptive hidden layers for mobile gesture recognition," in *AAAI*, 2018.

[30] C. Bauckhage, "K-means clustering is matrix factorization," *arXiv*, 2015.

[31] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *ICML*, 2000.

[32] T. Sherwood, E. Perelman, and G. Hamerly et al, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[33] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, 2008.

[34] G. Shafer and V. Vovk, "A tutorial on conformal prediction," *Journal of Machine Learning Research*, 2008.

[35] V. Balasubramanian *et al.*, *Conformal prediction for reliable machine learning: theory, adaptations and applications*, 2014.

[36] D. Volkhonskiy *et al.*, "Inductive conformal martingales for change-point detection," *arXiv*, 2017.

[37] M. Solutions, "Power monitor." [Online]. Available: https://www.msoon.com/high-voltage-power-monitor

[38] W. Ertel, "On the definition of speedup," in *International Conference on Parallel Architectures and Languages Europe*, 1994.

[39] R. M. Bell, "The bellkor 2008 solution to the netflix prize."

[40] C.-K. Hsieh *et al.*, "Collaborative metric learning," in *WWW*, 2017.

[41] A. Shye *et al.*, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Micro '09*.

[42] M. Dong and L. Zhong, "Chameleon: A color-adaptive web browser for mobile oled displays," *IEEE Transactions on Mobile Computing*, 2012.

[43] M. Mehrara *et al.*, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *HPCA '11*.

[44] A. Sehati and M. Ghaderi, "Energy-delay tradeoff for request bundling on smartphones," in *INFOCOM*, 2017.

[45] Y. Rogers *et al.*, "Why it's worth the hassle: The value of in-situ studies when designing ubicomp," in *Ubicomp*, 2007.

[46] D. Ferreira *et al.*, "Contextual experience sampling of mobile application micro-usage," in *MobileHCI*, 2014.

[47] S. He *et al.*, "Optimizing smartphone power consumption through dynamic resolution scaling," in *MobiCom*, 2015.

[48] B. Zhao *et al.*, "Energy-aware web browsing on smartphones," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[49] A. Zare *et al.*, "Hevc-compliant tile-based streaming of panoramic video for virtual reality applications," in *MM*, 2016.

[50] D. H. Bui *et al.*, "Rethinking energy-performance trade-off in mobile web page loading," in *MobiCom*, 2015.

[51] M. A. Hoque *et al.*, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Comput. Surv.*, 2015.

[52] B. Taylor *et al.*, "Adaptive optimization for opencl programs on embedded heterogeneous systems," in *LCTES*, 2017.

[53] Z. Wang *et al.*, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM TACO*, 2014.

[54] G. Tournavitis *et al.*, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," in *PLDI*, 2009.

[55] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *PPoPP*, 2009.

[56] ——, "Partitioning streaming parallelism for multi-cores: a machine learning based approach," in *PACT*, 2010.

[57] D. Grewe *et al.*, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *CGO*, 2013.

[58] Z. Wang and M. F. O'boyle, "Using machine learning to partition streaming programs," *ACM TACO*, 2013.

[59] Z. Wang *et al.*, "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems," *ACM TACO*, 2014.

[60] W. F. Ogilvie *et al.*, "Fast automatic heuristic construction using active learning," in *LCPC*, 2014.

[61] Y. Wen *et al.*, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms," in *HiPC*, 2014.

[62] C. Cummins *et al.*, "End-to-end deep learning of optimization heuristics," in *PACT*, 2017.

[63] W. F. Ogilvie *et al.*, "Minimizing the cost of iterative compilation with active learning," in *CGO*, 2017.

[64] S. Chen *et al.*, "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *HPCC*, 2018.

[65] P. Zhang, *et al.*, "Auto-tuning streamed applications on intel xeon phi," in *IPDPS*, 2018.

[66] C. Lindong *et al.*, "Optimizing sparse matrix-vector multiplications on an armv8-based many-core architecture," *IJPP*, 2018.

[67] V. Sanz Marco, B. Taylor, Z. Wang, and Y. Elkhatib, "Optimizing deep learning inference on embedded systems through adaptive model selection," *ACM Transactions on Embedded Computing*, 2019.

[68] D. Grewe *et al.*, "A workload-aware mapping approach for data-parallel programs," in *HiPEAC*, 2011.

[69] M. K. Emani *et al.*, "Smart, adaptive mapping of parallelism in the presence of external workload," in *CGO*, 2013.

[70] D. Grewe *et al.*, "Opencl task partitioning in the presence of gpu contention," in *LCPC*, 2013.

[71] V. S. Marco *et al.*, "Improving spark application throughput via memory aware task co-location: a mixture of experts approach," in *ACM/IFIP/USENIX Middleware Conference*, 2017.

[72] B. Taylor *et al.*, "Adaptive deep learning model selection on embedded systems," in *LCTES*, 2018.

[73] J. Zhang *et al.*, "Crosssense: Towards cross-site and large-scale wifi sensing," in *MobiCom*, 2018.

[74] R. Jordaney *et al.*, "Transcend: Detecting concept drift in malware classification models," in {*USENIX*} *Security*, 2017.