Network Delay-Aware Optimization for Web Browsing on Heterogeneous Mobile Platforms

Jie Ren†, Xiaoming Wang†, Feng Tian†, Hai Wang‡, Jie Zheng‡, Zheng Wang*
† *Shaanxi Normal University, China,* ‡*Northwest University, China,* **Lancaster University, UK*
†{*renjie, wangxm, tianfeng*}*@snnu.edu.cn,* ‡{*hwang, jzheng*}*@nwu.edu.cn,* **z.wang@lancaster.ac.uk*

***Abstract*—In this paper, we propose a machine learning based approach to predict which of the heterogeneous processors to use to render the web content and the operating frequencies of heterogeneous processors. We do so by first learning, *offline*, a set of predictive models for a range of networking environments. We then choose a learnt model at runtime to predict the optimal processor configuration. The prediction is based on the web content, the network status and the optimization goal. We obtain, on average, over 21% and 39% of improvement respectively for load time and energy consumption, when compared to two competitive optimisers that are specifically tuned for mobile web browsing.**

1. Introduction

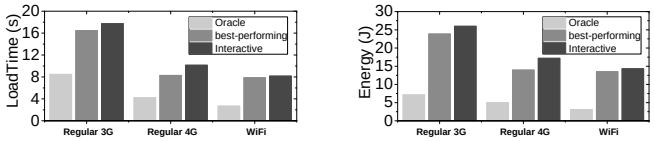
Web is a major information portal on mobile devices [**?**]. However, web browsing is poorly optimized and continues to consume a significant portion of battery power on mobile devices [**?**], [**?**], [**?**]. Heterogeneous multi-cores, such as the ARM big.LITTLE architecture [**?**], offer a new way for energy-efficient mobile computing. These platforms integrate multiple processor cores on the same system, where each processor is tuned for a certain class of workloads to meet a variety of user requirements. However, current mobile web browsers rely on the operating system to exploit the heterogeneous cores. Since the operating system has little knowledge of the web workload and how does the network affect web rendering, the decision made by the operating system is often sub-optimal. This leads to poor energy efficiency [**?**], draining the battery faster than necessary and irritating mobile users.

Rather than letting the operating system make all the scheduling decisions by passively observing the system's load, our work enables the browser to actively participate in decision making. Specifically, we want the browser to decide which heterogeneous core and the optimum processor frequencies to use to run the rendering engine. To effectively schedule the rendering process, the decisions must consider the web content, the optimization goal, and how the network affects the rendering process. Instead of developing a hand-crafted approach that requires expert insight into a specific computing and networking environment, we wish to develop an automatic technique that can be portable across environments. We achieve this by employing machine learning to automatically build predictors based on empirical observations gathered from a set of training web pages. The trained models are then used at runtime by the web browser to predict the optimum processor configuration for any *unseen* webpage.

The key contribution of this paper is a novel machine learning based web rendering scheduler that can leverage knowledge of the network and webpages to optimize mobile web browsing. We compare our approach against two state-of-art web browser schedulers [**?**], [**?**] on a representative ARM big.LITTLE mobile platform. Experimental results show that our approach outperforms the state-of-the-arts by delivering over 1.2x (up to 1.4x) improvement across evaluation metrics. Our techniques are generally applicable, as they are useful for not only web browsers but also a large number of mobile apps that are underpinned by web rendering techniques [**?**].

Table 1: The best-performing available governor

	Load time	Energy	EDP
Regular 3G	`performance`	`powersave`	`powersave`
Regular 4G	`performance`	`conservative`	`interactive`
WiFi	`interactive`	`ondemand`	`interactive`



(a) Load time (b) Energy consumption

Figure 1: The achieved total load time (a), energy consumption (b) when a user was browsing four news pages from `news.bbc.co.uk`. We show the results for `oracle`, the `best-performing` existing CPU frequency governor, and `interactive` in three typical networking environments. There is significant room for improvement.

2. Motivation

Consider a scenario for browsing four `BBC` news pages, starting from the home page of `news.bbc.co.uk`. Our evaluation device is an ARM big.LITTLE mobile platform with a Cortex-A15 (big) and a Cortex-A7 (little) processors.

Networking Environments. We consider three typical networking environments: Regular 3G, Regular 4G and WiFi (see Section **??** for more details). To ensure reproducible results, web requests and responses are deterministically replayed by the client and a web server respectively. The web server simulates the download speed and latency of a given network setting, and we record and deterministically replay the user interaction trace for each testing scenario.

Scheduling Strategies. We schedule the Chromium rendering engine (i.e., `CrRendererMain`) to run on either the big or the little core under different clock frequencies to find the best processor configuration per test case. We refer this best-found configuration as the `oracle` because it is the best performance we can get via CPU frequency scaling and task mapping. We use the `interactive` CPU frequency governor as the baseline, which is the default frequency governor on many mobile devices [**?**]. We also compare with the best-performing governor found from mainstream CPU

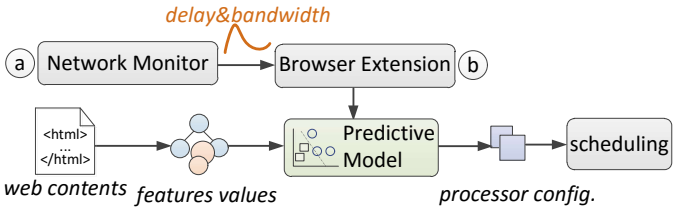


Figure 2: Overview of our approach. The network monitor evaluates the network bandwidth and delay to choose a model to predict the optimum processor configuration.

governors, including the `interactive` and other four strategies: `performance`, `conservative`, `ondemand` and `powersave`.

Motivation Results. Table **??** lists the best-performing governor chosen from the five existing CPU frequency governors, and Figure **??** summarizes the performance of each strategy for each optimization metric. While `interactive` gives the best `EDP` compared to other existing governors in a Regular 4G and a WiFi environments, it fails to deliver the best-available performance for load time and energy consumption. Furthermore, there is significant room for improvement for the best-performing existing governor when compared to the `oracle`. On average, the `oracle` outperforms the best-performing governor by 154.6%, 70.6% respectively for load time and energy consumption across networking environments. More importantly, the oracle processor configuration varies across web pages, networking environments and evaluation metrics – no single configuration consistently delivers the best-available performance.

3. Overview of our approach

As illustrated in Figure **??**, our approach consists of two components: (i) a network monitor running as an operating system service and (ii) a web browser extension. The network monitor measures the end to end delay and network bandwidths when downloading the webpage. The web browser extension determines the best processor configuration depending on the network environment and the web contents. We let the operating system to schedule other browser threads such as the input/output and the painting processes. At the heart of our web browser extension is a set of *off-line* learned predictive models. The predictor takes in a set of numerical values, or *features values*, which describes the essential characteristics of the webpage. It predicts which core to use to run the rendering process and at what frequency the heterogeneous processors should operate. The set of features used to describe the webpage is extracted from the web contents.

4. Predictive Modeling

Our models for processor configuration prediction are a set of Support Vector Machines (`SVMs`) [**?**]. We use the Radial basis kernel because it can model both linear and non-linear classification problems. We use the same methodology to learn all predictors for the target networking environments and optimization goals (i.e., load time, energy consumption, and `EDP`).

Table 2: Networking environment settings

	Uplink bandwidth	Downlink bandwidth	Delay
Regular 2G	50kbps	100kbps	1000ms
Good 2G	150kbps	250kbps	300ms
Regular 3G	300kbps	550kbps	500ms
Good 3G	1.5Mbps	5.0Mbps	100ms
Regular 4G	1.0Mbps	2.0Mbps	80ms
Good 4G	8.0Mbps	15.0Mbps	50ms
WiFi	15Mbps	30Mbps	5ms

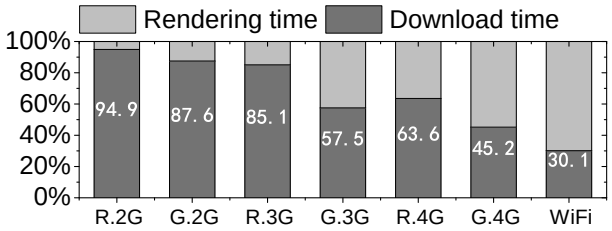


Figure 3: Webpage rendering time w.r.t. content download time when using the `interactive` governor.

4.1. Network Monitoring and Characterization

The communication network has a significant impact on the web rendering strategy. Table **??** lists the networking environments considered in this work. The settings and categorizations are based on the measurements given by an independent study [**?**]. Figure **??** shows the webpage rendering time with respect to the download time under each networking environment when using the `interactive` governor. The download time dominates the end to end turnaround time for a 2G and a Regular 3G environments; and by contrast, the rendering time accounts for most of the turnaround time for a Good 4G and a WiFi environments when the delay is small.

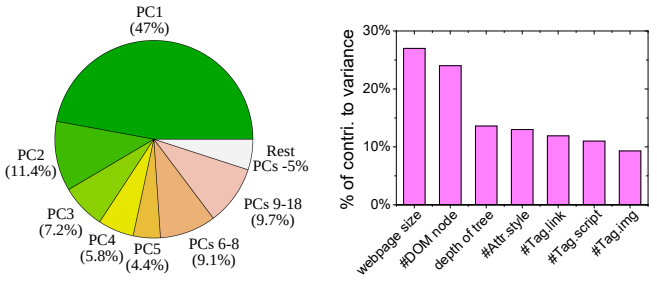
In this work, we learn a predictor per optimization goal for each of the seven networking environments. To determine which network environment the user is currently in, we develop a lightweight network monitor to measure the network bandwidths and delay between the web server and the device. The network monitor utilizes the communication link statistics that are readily available on commodity smartphones. Measured data are averaged over the measurement window. The measurements are then used to map the user's networking environment to one of the pre-defined settings in Table **??**, by finding which of the settings is closest to the measured values. The closeness or distance, d, is calculated using the following formula:

$$d = \alpha|db_m - db| + \beta|ub_m - ub| + \gamma|d_m - d| \qquad (1)$$

where db_m, ub_m, and d_m are the measured downlink bandwidth, upload bandwidth and delay respectively, db, ub, and d are the downlink bandwidth, upload bandwidth and delay of a network category, and α, β, γ are weights. The weights are automatically learned from the training data, with an averaged value of 0.3, 0.1 and 0.6 respectively for α, β, and γ.

4.2. Training the Predictor

The training process involves finding the best processor configuration and extracting feature values for each training webpage, and learn a model from the training data.



(a) Principal components (b) Top 7 most important features

Figure 4: The percentage of principal components (PCs) to the overall feature variance (a), and contributions of the seven most important raw features in the PCA space (b).

Generate Training Data. In this work, we used around 900 webpages to train a SVM predictor; we then evaluate the learnt model on the other 100 unseen webpages. These training webpages are selected from the landing page of the top 1000 hottest websites ranked by www.alexa.com. We use Netem [**?**], a Linux-based network enumerator, to emulate various networking environments to generate the training data. We exhaustively execute the rendering engine under different processor settings and record the optimal configuration for each optimization goal and each networking environment. We give each optimal configuration a unique label. For each webpage, we also extract values of a set of selected features.

Building The Model. The feature values together with the labeled processor configuration are supplied to a supervised learning algorithm [**?**]. The learning algorithm tries to find a correlation from the feature values to the optimal configuration and produces a SVM model per networking environment per optimization goal. Because we target two optimization metrics and seven networking environments, we have constructed 14 SVM models in total.

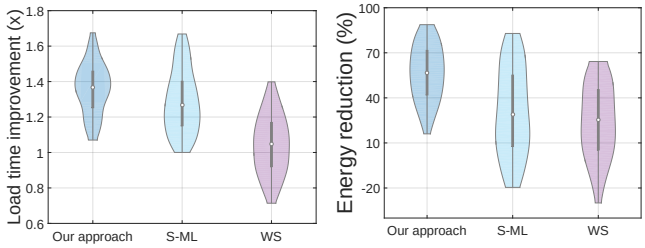
4.3. Web Features

One of the key aspects in building a successful predictor is finding the right features to characterize the input workload. In this work, we consider a set of features extracted from the web contents. These features are collected by our feature extraction pass. To gather the feature values, the feature extractor first obtains a reference for each DOM element by traversing the DOM tree and then uses the Chromium API, document.getElementsByID, to collect node information. We started from 214 raw features, including the number of DOM nodes, HTML tags and attributes of different types, and the depth of the DOM tree, etc. All these features can be collected at runtime from the browser. The types of the raw features are given in Table **??**. These features are selected based on our intuition and prior work [**?**], [**?**], [**?**]. It is important to note that the collected feature values are encoded to a vector of real values.

Feature Reduction. To improve the generalization ability of our models, i.e., reducing the likelihood of over-fitting on our training data, we reduce some features through applying Principal Component Analysis (PCA) [**?**] to the raw feature space. PCA transforms the original inputs into a set of principal components (PCs) that are linear combinations of the inputs. After applying PCA to the 214 raw features, we choose the top 18 principal components (PCs) which account for around 95% of the variance of the original feature space. We record the PCA transformation matrix and use it to transform the raw features of the new webpage to PCs during runtime deployment. Figure **??**a illustrates how much feature variance that each component accounts for. This figure shows that predictions can accurately draw upon a subset of aggregated feature values.

Table 3: Raw web feature categories

DOM Tree	#DOM nodes	depth of tree
	#each HTML tag	#each HTML attr.
Other	size of the webpage (Kilobytes)	



(a) Load Time (b) Energy Consumption

Figure 5: Violin plots showing the distribution for our approach, S−ML and WS in different network environments for three evaluation metrics: load time (a), energy reduction (b). The baseline is the best-performing Linux CPU governor. The thick line shows where 50% of the data lies. The white dot is the position of the median.

Feature Normalization. Before passing our features to a machine learning model we need to scale each of the features to a common range (between 0 and 1) in order to prevent the range of any single feature being a factor in its importance. Scaling features does not affect the distribution or variance of their values. To scale the features of a new webpage during deployment we record the minimum and maximum values of each feature in the training dataset, and use these to scale the corresponding features.

Feature Analysis. To understand the usefulness of each raw feature, we apply the Varimax rotation [**?**] to the PCA space. This technique quantifies the contribution of each feature to each PC. Figure **??**b shows the top 7 dominant features based on their contributions to the PCs. Features like the webpage size and the number of DOM nodes are most important, because they strongly correlate to the download time and the complexity of the webpage. Other features like the depth of the DOM tree, and the numbers of different attributes and tags, are also useful, because they determine how the webpage should be presented and how do they correlate to the rendering cost.

5. Experimental Results

The violin plot in Figure **??** compares our approach against two state-of-the-arts, S−ML and WS, across networking environments and webpages. The baseline is the best-performing Linux CPU governor found for each webpage. The width of each violin corresponds to the proportions of webpages with a certain improvement. The white dot

denotes the median value, while the thick black line shows where 50% of the data lies. The overhead of network monitoring, extracting features, prediction and configuring frequency is small. It is less than 7included in all our experimental results

On average, all approaches improve the baseline and the highest improvement is given by our approach. This confirms our hypothesis that knowing the characteristics of the web content can improve scheduling decisions. If we look at the bottom of each violin, we see that `WS` and `S-ML` can lead to poor performance in some cases. For example, `WS` gives worse performance for 40% of the webpages, with up to 30% slowdown for load time. `S-ML` delivers better performance when compared with `WS`, due to the more advanced modeling technique that it employs. However, `S-ML` also gives worse performance for 18% and 17% of the webpages for loadtime, energy respectively, and can consume up to 20% more energy than the baseline. The unstable performance of `WS` and `S-ML` is because they are unaware of the network status, and thus lead to poor performance in certain environments. By contrast, our approach never gives worse performance across networking environments and webpages. Finally, consider now the improvement distribution. There are more data points at the top of the diagram under our scheme. This means our approach delivers faster load time and greater reduction on energy and `EDP` when compared with `WS` and `S-ML`. Overall, our approach outperforms the competitive approaches with an average improvement of 21% and 39.2% respectively for load time and energy, and without ever giving worse performance when compared with the baseline. We also evaluate the performance of our techniques when web caching is enabled. The results show that our approach still outperforms the other two methods with similar improvements.

6. Related Work

Numerous techniques have been proposed to optimize web browsing, through e.g. prefetching [**?**] and caching [**?**] web contents, or re-constructing the browser workflow [**?**], [**?**] or the TCP protocol [**?**]. Most of the prior work target homogeneous systems and do not optimize across networking environments. The work presented by Zhu *et al.* [**?**] and prior work [**?**] were among the first attempts to optimize web browsing on heterogeneous mobile systems. Both approaches use statistical learning to estimate the optimal configuration for a given web page. However, they do not consider the impact of the networking environment, thus miss massive optimization opportunities. Bui *et al.* [**?**] proposed several web page rendering techniques to reduce energy consumption for mobile web browsing. Their approach uses analytical models to determine which processor core (big or little) to use to run the rendering process. The drawback of using an analytical model is that the model needs to be manually re-tuned for each individual platform to achieve the best performance. Our approach avoids the pitfall by developing an approach to automatically learn how to best schedule rendering process.

7. Conclusions

This paper has presented an automatic approach to optimize web rendering on heterogeneous mobile platforms, providing significant improvement over existing web-content-aware schedulers. We show that it is crucial to exploit the knowledge of the communication network and the web contents to make effective scheduling decisions. We address the problem by using machine learning to develop predictive models to predict which processor core to use to run the web rendering process and the optimum frequency of the processors. As a departure from prior work, our approach consider of the network status, web workloads and the optimization goals. Our techniques are implemented as an extension in the Chromium web browser and evaluated on a representative big.LITTLE heterogeneous mobile platform using the top 1000 hottest websites. Experimental results show that our approach achieves over 80% of the `oracle` performance, and outperforms the state-of-the-arts by 1.21x and 1.39x for load time and energy consumption.