Proteus: Network-aware Web Browsing on Heterogeneous Mobile Systems

Jie Ren, Xiaoming Wang
Shaanxi Normal University, China

Jianbin Fang
National University of Defense Technology, China

Yansong Feng
Peking University, China

Dongxiao Zhu
Wayne State University, USA

Zhunchen Luo
PLA Academy of Military Science, China

Jie Zheng
Northwest University, China

Zheng Wang
Lancaster University, U. K.

ABSTRACT

We present PROTEUS, a novel network-aware approach for optimizing web browsing on heterogeneous multi-core mobile systems. It employs machine learning techniques to predict which of the heterogeneous cores to use to render a given webpage and the operating frequencies of the processors. It achieves this by first learning *offline* a set of predictive models for a range of typical networking environments. A learnt model is then chosen at runtime to predict the optimal processor configuration, based on the web content, the network status and the optimization goal. We evaluate PROTEUS by implementing it into the open-source Chromium browser and testing it on two representative ARM big.LITTLE mobile multi-core platforms. We apply PROTEUS to the top 1,000 popular websites across seven typical network environments. PROTEUS achieves over 80% of best available performance. It obtains, on average, over 17% (up to 63%), 31% (up to 88%), and 30% (up to 91%) improvement respectively for load time, energy consumption and the energy delay product, when compared to two state-of-the-art approaches.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Computing methodologies → Parallel computing methodologies**;

KEYWORDS

Web browsing, Energy optimization, Heterogeneous multi-cores

ACM Reference Format:
Jie Ren, Xiaoming Wang, Jianbin Fang, Yansong Feng, Dongxiao Zhu, Zhunchen Luo, Jie Zheng, and Zheng Wang. 2018. Proteus: Network-aware Web Browsing on Heterogeneous Mobile Systems. In *CoNEXT '18: International Conference on emerging Networking EXperiments and Technologies, December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3281411.3281422

1 INTRODUCTION

Web is a major information portal on mobile systems [34]. However, web browsing is poorly optimized and consumes a significant portion of battery power on mobile devices [15, 19, 58]. Heterogeneous multi-cores, as representative by the ARM big.LITTLE architecture [1], have become the de facto hardware design for mobile platforms [49]. Such architectures integrate multiple processor cores, where each processor is tuned for a certain class of workloads to meet a variety of user requirements. To unlock the potential of heterogeneous multi-cores requires knowing which processor core to use and at what frequency the core should operate.

Current web browsers rely on the operating system to exploit the heterogeneous cores. Since the operating system (OS) has little knowledge of the web workload and how does the network affect web rendering, the decision made by the OS is often sub-optimal. This leads to poor energy efficiency [74], draining the battery faster than necessary and irritating mobile users. In this work, we ask the research question: "What advantages can a scheduler take when it knows the web workload and the impact of the networking environment?". In answer, we develop PROTEUS, a novel web browser task scheduler to exploit knowledge of the computing environment and web workloads to make better use of the underlying hardware.

The goal of PROTEUS is to choose the best processor (CPU and GPU) configuration for a given web workload for a specific network environment. We focus on processor scheduling because processors are the major energy consumer on mobile devices and their power consumption has continuously increased on recent processor generations [29]. Rather than relying on the OS to make all the scheduling decisions by passively observing the system's load, PROTEUS enables the browser to actively participate in decision making. Specifically, it enables the browser to decide which heterogeneous CPU core and the optimal CPU/GPU frequencies to use to run the rendering and painting processes. We show that the decision must be based on the web content, the optimization goal, and knowledge of how the network affects the rendering performance.

Instead of developing a hand-crafted approach that requires expert insight into a specific computing and networking environment, we put portability and adaptation at the core of PROTEUS. We

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281422

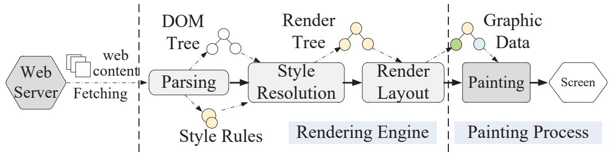


Figure 1: The processing procedure of Chromium.

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 optimal processor configuration for any *unseen* webpage. Such an approach avoids the pitfalls of using a hard-wired heuristics that require human modification every time the computing environment or hardware changes.

We implemented Proteus in the open-source Chromium web browser [6] and released it under an open-source license[1]. We evaluate Proteus by applying it to the top 1,000 popular websites ranked by `alexa.com` [4], including Facebook, Amazon, CNN, etc. We test Proteus under seven typical cellular and WiFi network settings and compare it against two state-of-the-art web browser schedulers [51, 75] on two distinct heterogeneous big.LITTLE mobile platforms: Odroid XU3 and Jetson TX2. We consider three metrics: load time, energy consumption and the energy delay product. Experimental results show that Proteus consistently outperforms prior methods across evaluation metrics and platforms.

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. Our results show that significant energy efficiency for heterogeneous mobile web browsing can be achieved if the scheduler is aware of the networking environment and the web workload. 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 [16].

2 BACKGROUND

2.1 Web Processing

Figure 1 illustrates how Chromium handles a webpage. The web contents, e.g., HTML pages, CSS styles, Javascripts and multimedia contents, are fetched by a network process. The downloaded content is processed by the *rendering engine process*. The rendering results are passed to the *painting process* to generate visualization data in the GPU buffer to display to the user. This pipeline of rendering and screen painting is called *content painting*. To render the web content, the rendering engine constructs a Document Object Model (DOM) tree where each node of the tree represents an individual HTML tag like <body> or <p>. CSS style rules that describe how the web contents should be presented are also parsed by the rendering engine to build the style rules. After parsing, styling information and the DOM tree are combined to build a render tree which is then used to compute the layout of each visible element. To display the rendered content, the painting process outputs the rendered data as pixels to the GPU buffer.

Table 1: The best-performing existing governor

	Load time		Energy		EDP	
	CPU	GPU	CPU	GPU	CPU	GPU
Regular 3G	Perf.	Default	powersave	Static	powersave	Booster
Regular 4G	Perf.	Default	conservative	Static	Inter.	Booster
WiFi	Inter.	Default	ondemand	Booster	Inter.	Booster

2.2 Problem Scope

Our work focuses on scheduling the time-consuming *rendering* and *painting* processes on heterogeneous mobile multi-cores. The goal is to develop a portable approach to automatically determine, for a given webpage in a network environment, the optimal processor configuration. A processor configuration consists of three parameters: (1) which heterogeneous CPU to use to run the rendering process, (2) what are the clock frequencies for the heterogeneous CPUs, and (3) the GPU frequency for running the painting process.

3 MOTIVATION EXAMPLE

Consider a scenario for browsing three BBC news pages, starting from the home page of `news.bbc.co.uk`. In the example, we assume that the user is an average reader who reads 280 words per minute [36] and would click to the next page after finishing reading the current one[2]. Our evaluation device in this experiment is Odroid XU3 (see Section 6.1), an ARM big.LITTLE mobile platform with a Cortex-A15 (big) and a Cortex-A7 (little) CPUs, and a Mali-T628 GPU.

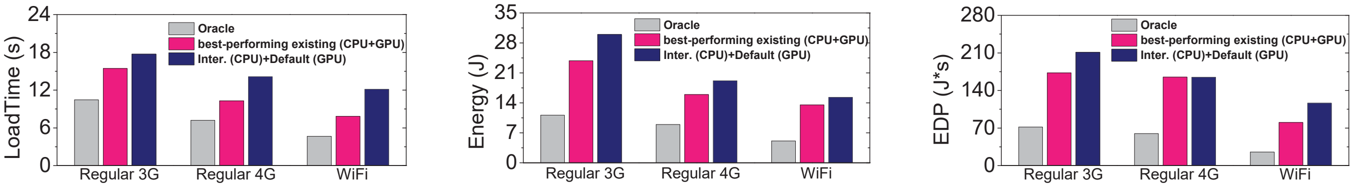
Networking Environments. We consider three typical networking environments (see Section 5.1 for more details): Regular 3G, Regular 4G and WiFi. To ensure reproducible results, web requests and responses are deterministically replayed by the client and a web server respectively. The web server simulates the download speed and latency of a network setting, and we record and deterministically replay the user interaction trace for each testing scenario. More details of our experimental setup can be found at Section 6.1.

Oracle Performance. We schedule the Chromium rendering engine (i.e., `CrRendererMain`) to run on either the big or the little CPU core under different clock frequencies. We also run the GPU painting process (i.e., `Chrome_InProcGpuThread`) under different GPU frequencies. We record the best processor configuration per test case per optimization target. We refer this best-found configuration as the `oracle` because it is the best performance we can get via processor frequency scaling and task mapping.

Scheduling Strategies. For rendering, we use the `interactive` CPU frequency governor as the baseline, which is the default frequency governor on many mobile devices [53]. We use the Android's default settings of `interactive`, i.e., it samples the CPU load every 80 ms, and raises the frequency if the CPU utilization is above 85%; after that, it waits for at least 20 ms before re-sampling the CPU to decide whether to lower or raise the frequency. We also compare to other four Linux-based CPU frequency governors: `performance`, `conservative`, `ondemand` and `powersave`. The GPU frequency is controlled by a GPU-architecture-dependent frequency

[1] Code is available at: https://github.com/Jiesourcecode/AMBER/.

[2] Note that the user's "think time" does not affect our approach as rendering will not start until the user has entered the website URL.



(a) Load time (b) Energy consumption (c) EDP

Figure 2: The total load time (a), energy consumption (b) and energy delay product (EDP) (c) when a user was browsing three news pages from news.bbc.co.uk. We show the results for oracle, the best-performing existing CPU and GPU frequency governors, and interactive (CPU) + Default (GPU) in three typical networking environments.

Table 2: Optimal configurations for BBC pages. The color box highlights which CPU the rendering process should run on while each color code represents a specific CPU frequency.

		Regular 3G			Regular 4G			WiFi		
		A15 (GHz)	A7 (GHz)	GPU (GHz)	A15 (GHz)	A7 (GHz)	GPU (GHz)	A15 (GHz)	A7 (GHz)	GPU (GHz)
Landing page	Load time	1.7	0.4	0.543	1.8	0.4	0.600	1.8	0.4	0.600
	Energy	0.4	0.8	0.400	0.4	0.8	0.400	0.8	0.4	0.543
	EDP	0.8	0.4	0.420	0.8	0.4	0.420	0.8	0.8	0.543
news page 1	Load time	1.6	0.4	0.543	1.6	0.4	0.600	1.7	0.4	0.600
	Energy consumption	0.8	0.4	0.420	0.8	0.4	0.420	0.8	0.4	0.420
	EDP	0.8	0.4	0.420	0.8	0.8	0.420	0.8	0.8	0.420
news page 2	Load time	1.6	0.4	0.543	1.7	0.4	0.600	1.8	0.4	0.600
	Energy consumption	0.4	0.4	0.350	0.4	0.8	0.400	0.8	0.8	0.420
	EDP	0.4	0.4	0.350	0.4	0.8	0.400	0.8	0.4	0.420

governor [28]. Here we consider all the three mainstream GPU frequency governors available on Odroid XU3: Default, Static and Booster; and we use Default as the baseline GPU frequency governor. We call the best-performing CPU and GPU frequency governor the *best-performing existing governor* thereafter.

Evaluation Metrics. In this work, we consider three *lower is better* metrics: *load time*, *energy consumption* and *energy delay product* (EDP) – calculated as *energy* × *load runtime* – a commonly used metric for quantifying the balance between energy consumption and load time [7, 24].

Motivation Results. Table 1 lists the best-performing existing governor for rendering and painting, and Figure 2 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. For painting, Default gives the best load time, Static saves the most energy, and Booster delivers the best EDP– the best GPU governor varies depending on which metric to be optimized. Furthermore, there is significant room for improvement for the best-performing *combination* of CPU and GPU governors when compared to the oracle. On average, the oracle outperforms the best-performing existing-governor combination by 34.2%, 53.1%, and 63.6% respectively for load time, energy consumption and EDP across networking environments. Table 2 presents the optimal configuration found by exhaustively trying all possible processor configurations. The core used for running the rendering process is highlighted using a color box, where each color code represents a specific CPU frequency. We note that the optimal frequency for the little core (A7) also vary even when it is not used for rendering. This is because the A7 core may be used to run other browser processes such as I/O threads. As can be seen from the table, the optimal processor configuration varies across web pages, networking environments and evaluation metrics – no single configuration consistently delivers the best-available performance.

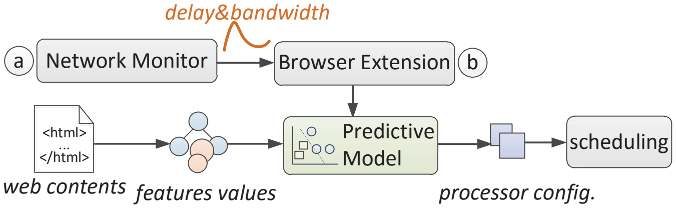


Figure 3: Overview of Proteus. The network monitor evaluates the network bandwidth and delay to choose a model to predict the optimal processor configuration.

Lessons Learned. This example shows that the current mainstream CPU frequency governors are ill-suited for mobile web browsing and the best processor configuration depends on the network and the optimization goal. Later in Section 7.1, we will show that similar results are also observed for other webpages. Clearly, there is a need for a better scheduler that can adapt to the webpage workload, the network environment and the optimization goal. In the remainder of this paper, we describe such an approach based on machine learning.

Table 3: Network 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

4 OVERVIEW OF PROTEUS

As illustrated in Figure 3, Proteus 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 processes.

At the heart of our web browser extension is a set of *off-line* learned predictive models, each targets a specific networking environment and a user specified optimization goal. The network status reported by the network monitor is used to choose a predictor. After training, the learnt models can then be used for any *unseen* webpage. The predictor takes in a set of numerical values, or *features values*, which describes the essential characteristics of the webpage. It predicts what processor configuration to use to run the rendering and painting processes on the the heterogeneous multi-core platform. The set of features used to describe the webpage is extracted from the web contents. This is detailed in Section 5.3. In Section 7.3.6, we show that a simple model like a decision tree or linear regression would fail to capture the non-linear behaviors among webpage features and the optimizations, and hence, a more sophisticated predictive model is required.

5 PREDICTIVE MODELING

Our goal for choosing which of a set of processor configurations to use can be naturally modeled as a classification problem. Our predictive models are a set of Support Vector Machines (SVMs) [61]. 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 target network environment and optimization goal (i.e., load time, energy consumption, or EDP) per hardware platform. We choose SVMs because they deliver better performance than alternative modeling techniques (see Section 7.3.6).

Building and using a predictive model follows the well-known 4-step process for supervised learning: (1) modeling the problem domain, (2) generating training data (3) learning a predictive model and (4) using the predictor. These steps are described as follows.

5.1 Network Monitoring and Characterization

The communication network has a significant impact on the web rendering and painting strategy. Intuitively, if a user has access to a fast network, he/she would typically expect quick response time for webpage rendering; on the other hand, if the network is slow, operating the processor at a high frequency would be unnecessarily because the content downloading would dominate the turnaround time and in this scenario the bottleneck is the I/O not the CPU.

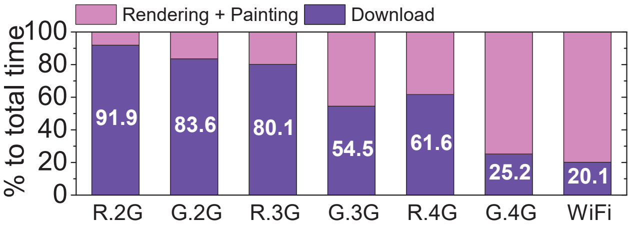


Figure 4: Webpage rendering and painting time w.r.t. content download time when using the `interactive` (CPU) and the `Default` (GPU) governors on Odroid XU3.

Table 3 lists the network environments considered in this work. The settings and categorizations are based on the measurements given by an independent study [3]. Figure 4 shows the webpage rendering time with respect to the download time in each 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. Our framework allows new predictors to be added to target different environments and no retraining is required for existing predictors. Because the process of model training and data collection can be performed automatically, our approach can be easily ported to a new hardware platform or network environment.

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, i.e., between the browser establishes the connection and making a prediction. The measurements are then used to map the user’s network environment to one of the pre-defined settings in Table 3, 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.

5.2 Training the Predictor

Figure 5 depicts the process of using training webpages to build a SVM classifier for an optimization target under a network environment. Training involves finding the best processor configuration and extracting feature values for each training webpage, and learn a model from the training data.

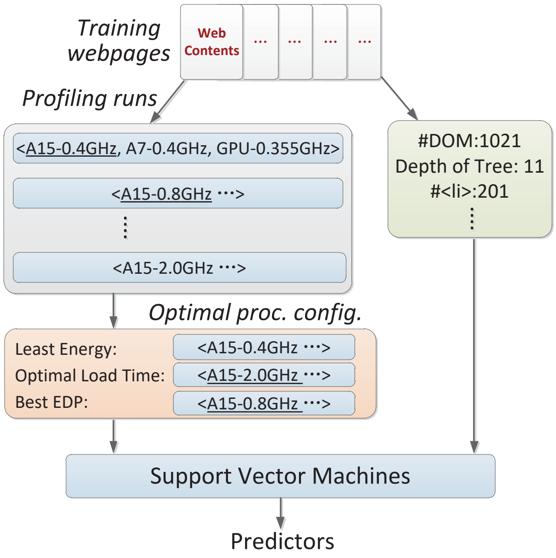


Figure 5: Learning predictive models using training data collected from a target network environment.

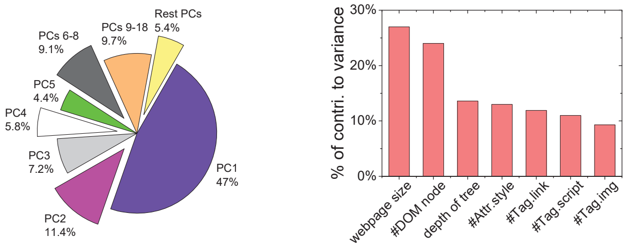
Generate Training Data. In this work, we used around 900 webpages to train a SVM predictor; we then evaluate the learnt model on the other 100 unseen webpages. These training webpages are selected from the landing page of the top 1000 hottest websites ranked by www.alexa.com (see Section 6.2). We use Netem [30], a Linux-based network emulator, to emulate various networking environments to generate the training data (see also Section 6.1). We exhaustively execute the rendering engine and painting process under different processor settings and record the optimal configuration for each optimization goal and each network environment. We then assign each optimal configuration a unique label. For each webpage, we also extract values of a set of selected features and store the values in a fixed vector (see Section 5.3).

Building The Model. The feature values together with the labeled processor configuration are supplied to a supervised learning algorithm [35]. The learning algorithm tries to find a correlation from the feature values to the optimal configuration and produces a SVM model per network environment per optimization goal. Because we target three optimization metrics and seven networking environments, we have constructed 21 SVM models in total for a given platform. An alternative is to have a single model for all optimization metrics and networking environments. However, this strategy requires retraining the model when targeting a new metric or environment and thus incurs extra training overheads.

Training Cost. The training time of our approach is dominated by generating the training data. In this work, it takes less than a week to collect all the training data. 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. Since training is only performed once at the factory, it is a *one-off* cost.

5.3 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.



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

Figure 6: 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).

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 the parsing time from the browser. The types of the raw features are given in Table 4. Some of these features are selected based on our intuition which may be important for our problem, while others are chosen based on prior work [9, 43, 51]. The collected feature values are encoded to a vector of real values. One of the advantages of our web features is that the feature values are obtained at the very beginning of the loading process, which gives enough time for runtime optimization.

It is important to note that our chosen features are independent of the underlying web browser and hardware platform. This is essential for making sure our approach are portable across platforms. While we do not use hardware- or browser-specific features, the platform characteristics, such as the processing capability and input/output latency, are implicitly captured from the training data.

Feature Reduction. The time spent in making a prediction is negligible in comparison to the overhead of feature extraction, therefore by reducing our feature count we can decrease the overhead of our predictive models. Moreover, by reducing the number of features we are also improving the generalization ability of our models, i.e., reducing the likelihood of over-fitting on our training data. Feature reduction is automatically performed through applying Principal Component Analysis (PCA) [20] 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 6a 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.

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

Table 4: Raw web feature categories

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

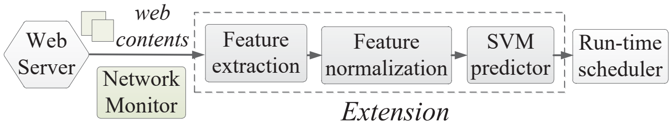


Figure 7: Runtime deployment of Proteus.

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. We also simply truncate a feature value to 0 or 1 if it is respectively smaller or greater than the minimum or maximum values seen in the training dataset.

Feature Analysis. To understand the usefulness of each raw feature, we apply the Varimax rotation [40] to the PCA space. This technique quantifies the contribution of each feature to each PC. Figure 6b 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 with 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. The advantage of our feature selection process is that it automatically determines what features are useful when targeting a new hardware platform where the relative cost of page rendering and the importance of features may change.

5.4 Runtime Deployment

Once we have built the predictive models described above, we can use them for any *new, unseen* webpage. Figure 7 illustrates the steps of runtime prediction and task scheduling. The network monitor reports the network bandwidths and delay, which are used to determine the runtime status. The web browser then selects a predictor to use based on the network status and the optimization goal. During the parsing stage, which takes less than 1% of the total rendering time [41], the feature extractor firstly extracts and normalizes the feature values. Next, the selected predictive model predicts the optimal processor frequency based on the feature values. The prediction is then passed to the runtime scheduler to perform task scheduling and hardware configuration. The overhead of network monitoring, extracting features, prediction and configuring frequency is small. It is less than 1% of the turnaround time (see also Section 7.3.3), *which is included in all our experimental results*.

As the DOM tree is constructed incrementally by the parser, it can change throughout the duration of rendering. To make sure that our approach can adapt to the change of available information, re-prediction and rescheduling will be triggered if the DOM tree is significantly different from the one used for the last prediction. The difference is calculated by counting the number of DOM nodes between the previous and the current DOM trees. If the difference is greater than 30%, we will make a new prediction using feature values extracted from the current DOM tree. We note that this threshold is empirically determined from training data. We have observed that our initial prediction often remains unchanged, so rescheduling and reconfiguration rarely happened in our experiments.

Table 5: None-zero feature values for Google search ($p1$), the result page ($p2$) and the target website ($p3$).

Feature	Raw value			Normalized value		
	$p1$	$p2$	$p3$	$p1$	$p2$	$p3$
#DOM nodes	397	1292	4798	0.049	0.163	0.611
depth of tree	13	21	23	0.416	0.750	0.833
#img	3	5	169	0.004	0.007	0.256
#li	19	76	799	0.011	0.046	0.490
#link	2	8	3	0.026	0.106	0.04
#script	13	79	54	0.099	0.603	0.412
#href	46	155	2044	0.022	0.075	0.99
#src	3	21	84	0.006	0.043	0.167
#content	2	23	11	0.039	0.450	0.215

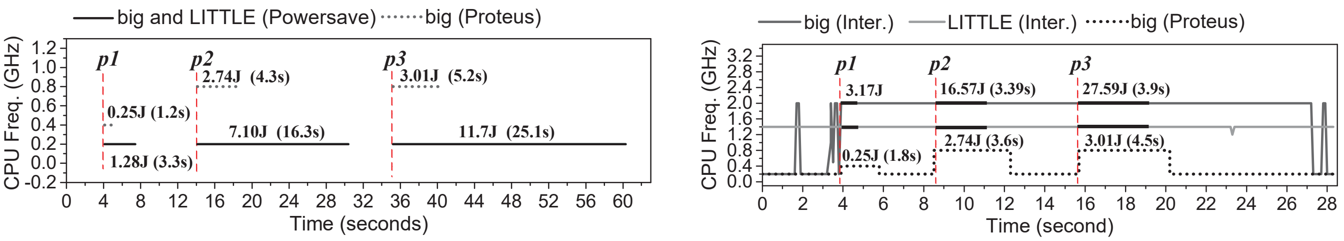
5.5 Working Example

As an example, we now consider a scenario where a user conducts a search on Google to look for an online service. There are three webpages to be rendered in this process: the Google search page, the search result page, and the target website, which are denoted as *p1*, *p2* and *p3* respectively. Here we assume the user uses a Regular 3G network and wants to retrieve the information with minimum energy usage by informing the system via e.g., choosing the battery saver mode on Android.

For this example, an energy-tuned predictor for a Regular 3G network is chosen. The feature extractor gathers the raw feature values from the DOM tree after the browser starts parsing the web content. The feature values are normalized and projected into the PCA space as described in Section 5.3. Table 5 lists some of the non-zero raw feature values for the three webpages, before and after normalization. These processed feature values will be fed into the selected SVM model. The model outputs a label (<A15 - 0.4, 0.4, GPU - 0.355> for Google search), indicating the optimal configuration is to run the rendering process on the big core and the clock frequency of the little and big cores should be set to 400 MHz, and the painting proocess on the GPU with 355 MHz. This prediction is indeed the ideal processor configuration. Finally, the processor configuration is communicated to the runtime scheduler to configure the hardware platform. For the other two webpages, our model also gives the optimal configuration, <A15 - 0.8, 0.8, GPU - 0.42>.

Figure 8a compares the powersave CPU frequency governor with Proteus. This strategy runs all cores at the lowest frequency, 200MHz, aiming to minimize the system's power consumption. However, running the processors at this frequency prolongs the page load time, which leads to over 1.59x (up to 4.12x) more energy consumption than Proteus.

In contrast to the fixed strategy used by powersave, the widely used interactive governor dynamically adjusts the processor frequency according to the user activities. From Figure 8b, we see that interactive raises the big core frequency as soon as the browser starts fetching $p1$. After that all cores stay on the highest frequency



(a) Compare with the powersave governor

(b) Compare with the interactive governor

Figure 8: The selected processor and CPU frequencies when rendering Google search ($p1$), the search result page ($p2$), and the target website ($p3$). We compare Proteus against powersave (a) and interactive (b) in a regular 3G environment.

Table 6: Hardware platforms

	Odroid XU3	Jetson TX2
big CPU	32bit quad-core Cortex-A15 @ 2GHz	64bit quad-core Cortex-A57 @ 2.0 GHz
LITTLE CPU	32bit quad-core Cortex-A7 @ 1.4GHz	64bit dual-core Denver2 @ 2 GHz
GPU	8-core Mali-T628 @ 600MHz	256-core NVIDIA Pascal @ 1.3GHz

until a few seconds after the third webpage has been completely rendered. While interactive can choose CPU frequencies from the entire spectrum, it mostly focuses on the highest and the lowest frequencies. By contrast, Proteus dynamically adjusts the processor frequency according to the web content and browsing activities. It chooses to operate the processors at 400MHz for the relatively simple $p1$ page that has the smallest number of DOM nodes, and then raises the frequency up to 800MHz for the next two more complex pages. As a result, Proteus reduces the energy consumption by 87% at the cost of 22% slower when compared with interactive. Considering the goal is to minimize the energy consumption, Proteus outperforms interactive on this task.

6 EXPERIMENTAL SETUP

6.1 Hardware and Software Platforms

Evaluation Platform. To demonstrate the performance portability of Proteus, we evaluate it on two distinct mobile platforms, Odroid XU3 and Jetson TX2. Table 6 gives detailed information of both platforms. We chose these platforms as they are a representative big.LITTLE embedded architecture and has on-board energy sensors for power measurement. Both systems run Ubuntu 16.04 with the big.LITTLE enabled scheduler[3]. We used the on board energy sensors and external power monitor to measure the energy of the *entire* system. These sensors have been checked against external power measurement instruments and proven to be accurate in prior work [33]. We implemented our approach in Google Chromium (ver. 64.0) which is compiled using the gcc compiler (ver. 7.2).

Networking Environments. For reproducibility, we evaluate all schemes in a controlled environment. Specifically, we use a Linux server to record and replay the server responses through the Web Page Replay tool [2]. Our mobile test board and the web server communicate through WiFi, but we use Netem [30] to control the network delay and server bandwidth to simulate the seven networking environments defined in Table 3. We add 30% of variances (which follow a normal distribution) to the bandwidths, delay and packet loss to simulate a dynamic network environment. Note that we ensure that the network variances are the same during the replay of a test page. We also measure the difference of power between the WiFi and the cellular interfaces, and use this to calibrate the energy consumption in cellular environments. Finally, unless stated otherwise, we disabled the browser’s cache to provide a fair comparison across different methods (see also Section 7.2).

Workloads. We used the landing page of the top 1,000 hottest websites from www.alexa.com. We include both the mobile and the desktop versions of the websites, because many mobile users still prefer the desktop-version for their richer content and experience [13]. Figure 9 shows the CDF of the number of DOM nodes, web content sizes and load time when using the interactive governor in a WiFi environment. The DOM node and webpage sizes range from small (4 DOM nodes and 40 KB) to large (over 8,000 DOM nodes and 6 MB), and the load time is between 0.13 second and 15.4 seconds, suggesting that our test data cover a diverse set of web contents.

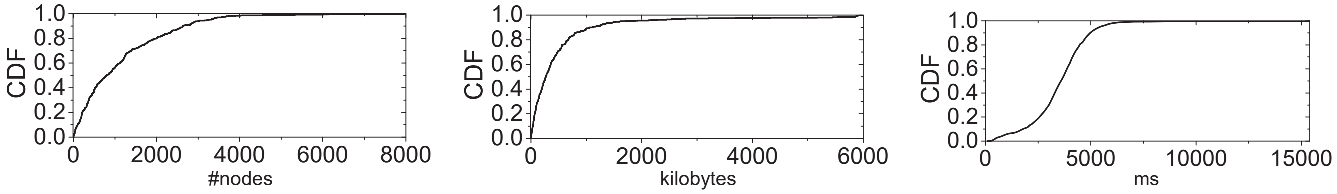
6.2 Evaluation Methodology

Model Evaluation. We use *10-fold cross-validation* to evaluate our machine learning models. Specifically, we partition the webpages into 10 sets where each set contains 100 webpages. We retain one set as the validation data for testing our model, and the remaining 9 sets are used as training data to train the model. We repeat this process 10 times (folds), with each of the 10 sets used exactly once as the validation data. We then report the averaged accuracy achieved across the 10 validation sets. This is a standard evaluation methodology, providing an estimate of the generalization ability of a machine-learning model in predicting *unseen* data.

Existing Frequency Governors. We compare Proteus against existing CPU and GPU frequency governors. Specifically, we consider five widely used CPU governors: interactive, powersave, performance, conservative, and ondemand. For GPUs, we consider three purpose-built governors for the ARM Mali GPU (Odroid Xu3): Default, Static and Booster, and three others for the NVIDIA Pascal GPU (Jetson TX2): nvhost_podgov, simple_ondemand and userspace. We use interactive as the baseline CPU governor, and Default and nvhost_podgov as the baseline GPU governor on Odroid XU3 and Jetson TX2, respectively.

Competitive Approaches. We compare Proteus with two state-of-the-art works: a web-aware scheduling mechanism (termed as

[3] Because Chromium for Android does not support extensions, we implemented Proteus on the Linux version that shares the same code base as the Android Chromium. We stress that our techniques can be built into the Chromium browser itself and is thus applicable to Android systems.



(a) #DOM nodes (b) Web page size (c) Load time (ms)

Figure 9: The CDF of #DOM nodes (a), webpage size (b), and load time when using interactive in a WiFi network (c).

WS) [75] and a machine learning based web browser scheduling scheme (termed as S-ML) [51]. WS uses a regression model to estimate webpage load time and energy consumption under different processor configurations. The model is then used as a cost function to find the best configuration by enumerating all possible configurations. S-ML also develops a machine learning classifier to predict the optimal processor configuration, but it assumes that all the webpages have been pre-downloaded and ignores the impact of the dynamic network environments. We train WS and S-ML using the same training dataset as the one we used to train our models in a WiFi environment (which is the networking environment used by both methods for collecting training data)

Performance Report. We report the *geometric mean* across evaluation scenarios. Compared to the arithmetic mean, it can better minimize the impact of performance outliers – which could make our results look better than they are [22]. To collect run-time and energy consumption, we run each model on each input repeatedly until the 95% confidence bound per model per input is smaller than 5%. For load time, we instrumented Chromium to measure the wall clock time between the Navigation_Start and the Load_Event_End events. We excluded the time spent on browser bootstrap and shut down. To measure the energy consumption, we developed a lightweight runtime to take readings from the on-board energy sensors at a frequency of 100 samples per second. We then matched the energy readings against the time stamps of webpage rendering to calculate the energy consumption.

7 EXPERIMENTAL RESULTS

Highlights of our evaluation are as follows:

- Proteus consistently outperforms the existing Linux-based governors across networking environments, optimization goals, and hardware platforms. See Section 7.1;
- Proteus gives better and more stable performance compared to state-of-the-art web-aware schedulers (Section 7.2);
- We thoroughly evaluate Proteus and provide detailed analysis on its working mechanisms (Section 7.3).

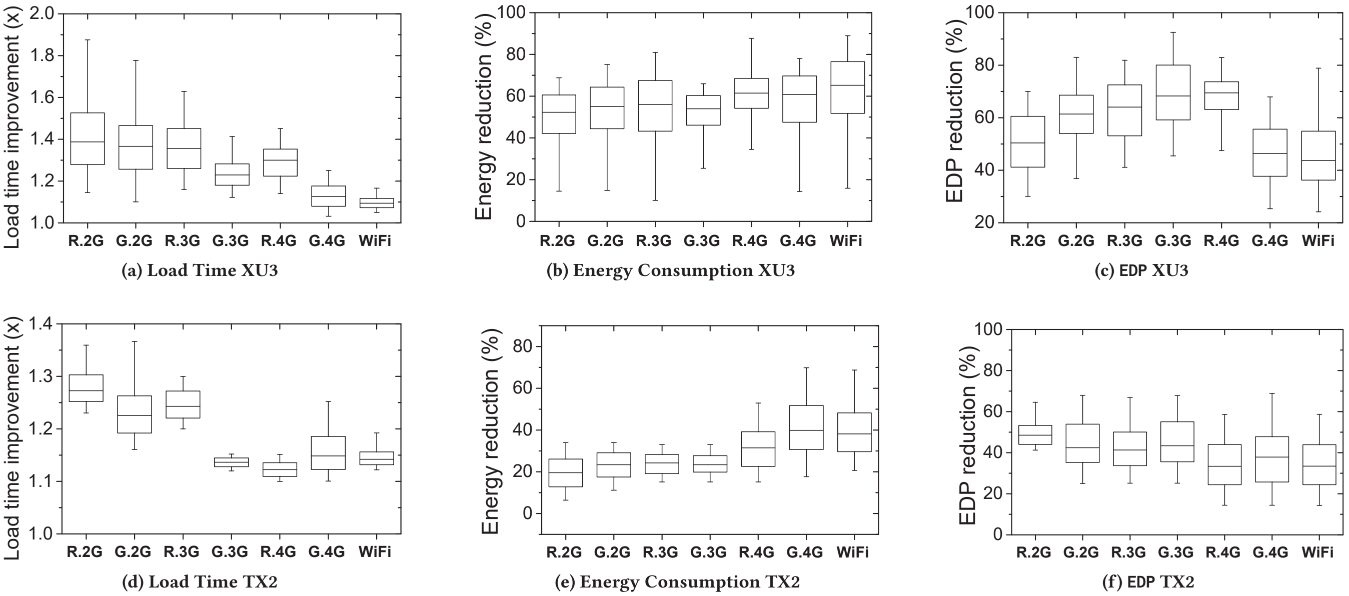
7.1 Overall Results

The box-plot in Figure 10 depicts the improvements of Proteus over the best-performing Linux-based governor. The min-max bars show the range of improvements achieved across webpages.

Load Time. Figure 10a and Figure 10d show the improvement of load time on Odroid XU3 and Jetson TX2, respectively. For this metric, the performance governor is the best-performing Linux governor for most of the test cases. Proteus delivers significantly better performance in slow networking environments like a 2G or a 3G network on both of two platforms, offering at least 11% quicker turnaround time. A slow network prolongs the webpage download time; and as a result, running the CPU and GPU at the highest frequency is not beneficial as the CPU sits idle for most of the time waiting for I/O, and the GPU waits to paint the rendered graphic data from CPU. Such a strategy would trigger frequent CPU [5] or GPU throttling [48], i.e. the hardware thermal manager would drop the clock frequency from 2GHz to 1.5GHz (or a lower frequency) to prevent the chip from overheating. Proteus learns from empirical observations that it is better to run the CPU and GPU at a slightly lower frequency, e.g., 1.8 GHz instead of 2 GHz, so that the CPU can operate on, on average, a higher frequency over the rendering period because of the less frequent CPU throttling. There is less improvement in a fast network like a WiFi environment. In such an environment, the download speed is no longer a bottleneck and running the CPU at a high frequency is often beneficial. Nonetheless, Proteus outperforms the best-performing Linux governor by 1.20x (2.5 seconds) on average (up to 1.87x, 6.2 seconds) across network environments and never gives worse performance.

Energy Consumption. Figures 10b and 10e compare our approach against other frequency governors in scenarios where low battery consumption is the first priority. In this case, powersave is the best-performing Linux governor in 2G and a Regular 3G environments, while conservative and ondemand are the best-performing Linux policies in a faster environment (Good 3G onwards). On average, Proteus outperforms the best-performing Linux governor by using less than 31% to 55% (up to 88%, 19 joules) energy consumption across networking environments. It is worth mentioning that Proteus never consumes more energy compared to other Linux governors, because it correctly selects the optimal (or near optimal) frequency and the best core to run the rendering process.

EDP. Figure 10c shows the results for EDP. A low EDP value means that energy consumption is reduced at the cost of little impact on the response time. Proteus successfully cuts down the EDP across networking environments. We observe significant improvement is available in a 3G and a Regular 4G environments, where Proteus gives over 60% (212 J*s) and 30% (182 J*s) reduction on EDP for Odroid XU3 and Jetson TX2, respectively. Proteus also reduces the EDP by over 30% in other networking environments. Once again, Proteus outperforms the best-performing Linux governor for all the test cases.



(a) Load Time XU3

(b) Energy Consumption XU3

(c) EDP XU3

(d) Load Time TX2

(e) Energy Consumption TX2

(f) EDP TX2

Figure 10: Improvement achieved by Proteus over the best-performing Linux CPU governor for load time, energy reduction and EDP on Odroid XU3 and Jetson TX2. The min-max bars show the range of performance improvement across webpages.

7.2 Compare to Competitive Approaches

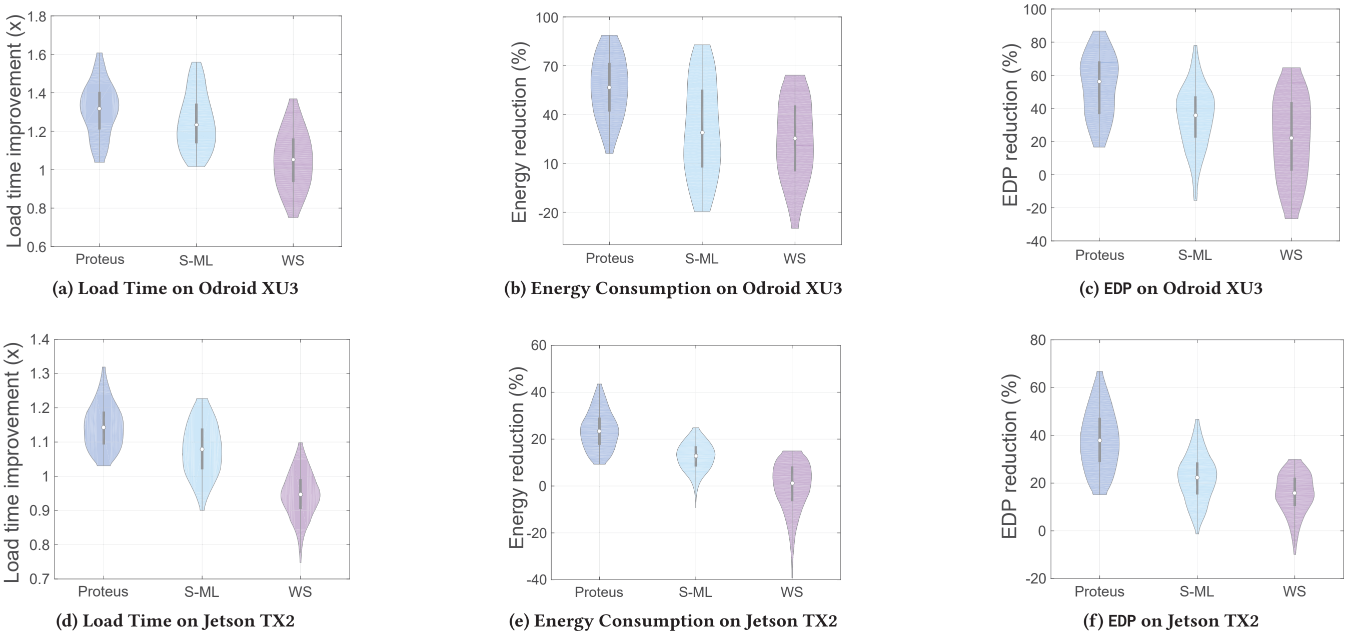
The violin plots in Figure 11 show the performance distributions achieved by Proteus, S-ML and WS across network environments and webpages. The baseline is the best-performing Linux CPU/GPU governor found for each webpage. The width of each violin corresponds to the proportions of webpages with a certain improvement. The white dot denotes the median value, while the thick black line shows where 50% of the data lies.

On average, all approaches improve the baseline and the highest improvement is given by Proteus. 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, 25% more energy and 30% worse for EDP on Odroid XU3. 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 and 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 Proteus delivers faster load time and greater reduction on energy and EDP when compared with WS and S-ML. Overall, Proteus outperforms the competitive approaches on two representative mobile platforms, Odroid XU3 and Jetson TX2, with an average improvement of 27.2% and 14.4% for load time, reduces the total energy consumption by 55.9% and 23.7% and improves the 56.4%, 38.1% for EDP, respectively. Overall, Proteus consistently outperforms WS and S-ML on both platforms and all metrics, and never delivers worse performance when compared with the baseline.

7.3 Model Analysis

7.3.1 Optimal configurations. Figure 12 shows the distribution for the most-frequently used optimal processor configurations found through exhaustive search. Here, we use the notation <**CPU render core - bigfreq, littlefreq, GPU-freq**> to denote a processor configuration. For example, <A15 - 1.6, 0.4, GPU-1.1> means that the rendering process running on the A15 core (big core) at 1.6GHz and the A7 core (little core) runs at 400MHz, and the painting process running on the GPU at 1.1GHz.

As can be seen from Figures 12a and 12d, when optimizing for load time, the rendering engine should run on the big core (A15 or A57) to provide high performance. However, the optimal frequency varies across networking environments and we see the change of distribution in frequencies when moving from a slow network to a fast one. For instance, on Odroid XU3, while it is unprofitable to run the A15 core at 1.9GHz and GPU at 0.6GHz in a slow network, it is the desired frequency for 68% of the webpages in a WiFi environment. When optimizing for energy consumption (Figures 12b and 12e) and EDP (Figures 12c and 12f), it can be beneficial to run the rendering process on the energy-tuned core (A7 or D2). For example, in a 2G environment, running the rendering process on the A7 core with a frequency of 400MHz or 800MHz benefits up to 46% of webpages on Odroid XU3, although the distribution changes across networks and optimization metrics. If we compare the distributions across networks and metrics, we find that the best core for running the rendering process and the frequency varies across



(a) Load Time on Odroid XU3

(b) Energy Consumption on Odroid XU3

(c) EDP on Odroid XU3

(d) Load Time on Jetson TX2

(e) Energy Consumption on Jetson TX2

(f) EDP on Jetson TX2

Figure 11: Violin plots showing the distribution for Proteus, S-ML and WS in different network environments for three evaluation metrics on Odroid XU3 and Jetson TX2. The baseline is the best-performing Linux-based CPU/GPU governor. The thick line shows where 50% of the data lies. The white dot is the position of the median. Proteus delivers the best and most stable performance across testing scenarios.

networking environments, webpages and optimization goals. The results reinforce our claim that the scheduling policy must be aware of the network, web contents and optimization target.

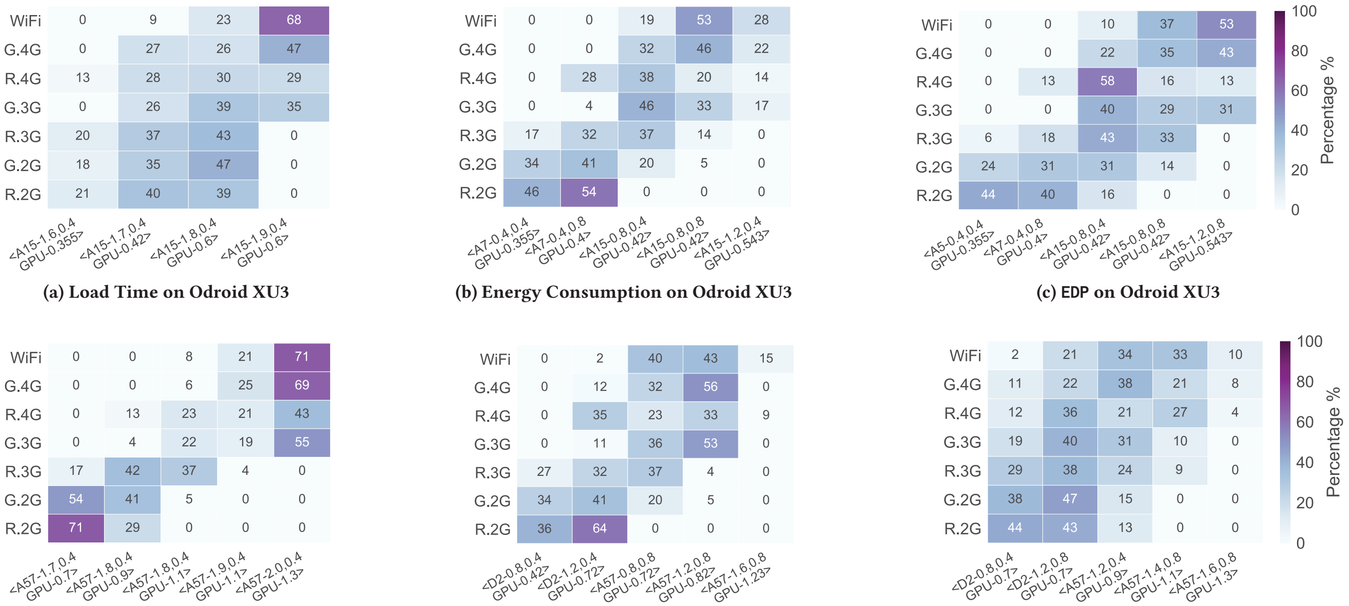
7.3.2 Feature Importance. Figure 13 shows a Hinton diagram that illustrates some of the most important features that have an impact on the energy consumption models. Here the larger the box, the more significantly a particular feature contributes to the prediction accuracy. The x-axis denotes the features and the y-axis denotes the models for the seven networking environments. The importance is calculated through the information gain ratio. It can be observed that HTML tags and attributes (e.g. webpage size, #DOM nodes , DOM tree depth) and style rules are important when determining the processor configurations for all networking environments. We can also see such features play an more important role for 2G and regular 3G than others. Other features are extremely important for some networks (such as the number HTML tags of `<Tag.script>` and `<Tag.li>`) are important for WiFi, 4G and good 3G, but less important for others. This diagram illustrates the need for a distinct model for each optimization goal and how important it is to have an automatic technique to construct such models.

7.3.3 Overhead breakdown . Figure 14 shows the overhead of Proteus (which is already included in our experimental results). Proteus introduces little overhead to the end to end turnaround time and energy consumption, less than 1% and 3% respectively. The majority of the time and energy are spent on network monitoring for measuring the network delay and bandwidths. The overhead incurred by the browser extension and the runtime scheduler, which includes task migration, feature extraction, making prediction and setting processor frequencies, is less than 0.3%, with task migration (around 10ms) accounts for most of the overhead. As can be seen from the better aforementioned results, the overhead of our approach can be amortized by the improved performance.

7.3.4 Oracle performance. Figure 15 compares Proteus with the `oracle` predictor, showing how close our approach is to the theoretically perfect solution. Our approach achieves 82%, 92% and 90% of the `oracle` performance for load time, energy consumption, and EDP respectively. Overall, the performance of Proteus is not far from the `oracle`.

7.3.5 Prediction accuracy. Proteus gives correct predictions for 85.1%, 90.1% and 91.2% of the webpages for load time, energy consumption and EDP respectively. For those webpages that Proteus does not give the best configuration, the resultant performance is not far from the optimal. We believe the accuracy of our approach can be improved by using more training examples, which in turn also permits to use a richer set of features.

7.3.6 Alternative modeling techniques. Figure 16 shows the performance achieved by our SVM-based approach and five widely used classification techniques with respect to the `oracle` performance. The alternative classifiers are: Multi-layer Perceptron (MLP), K-Nearest Neighbours (KNN), Artificial Neural Networks (ANN), Logistic Regression (LR), and Naïve Bayes (NB). Each of the alternative modeling techniques were trained and evaluated by using the same method and training data as our model. Proteus outperforms all other alternative techniques for every optimization metric. It is



(a) Load Time on Odroid XU3

(b) Energy Consumption on Odroid XU3

(c) EDP on Odroid XU3

(d) Load Time on Jetson TX2

(e) Energy Consumption on Jetson TX2

(f) EDP on Jetson TX2

Figure 12: Distributions of major optimal process configurations for load time, energy consumption and EDP. The distribution of optimal configuration changes across environments, showing the need of an adaptive scheme.

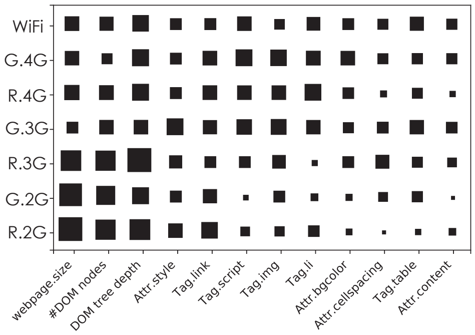


Figure 13: A Hinton diagram shows the importance of the selected web feature to the prediction accuracy under different networks. The larger the box, the more likely a feature affects the prediction accuracy of the respective model.

worth noting that the performance of these competitive modeling techniques may improve if there are more training examples to support the use of a larger set of features. However, we found that SVMs perform well on the training data we have.

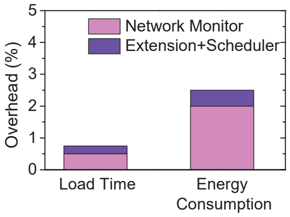


Figure 14: Breakdown of runtime overhead. Proteus incurs little runtime overhead.

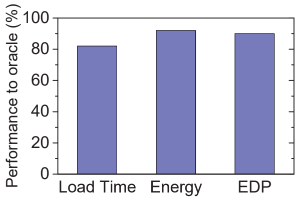


Figure 15: Performance of Proteus w.r.t. oracle. Proteus delivers over 80% of the oracle performance.

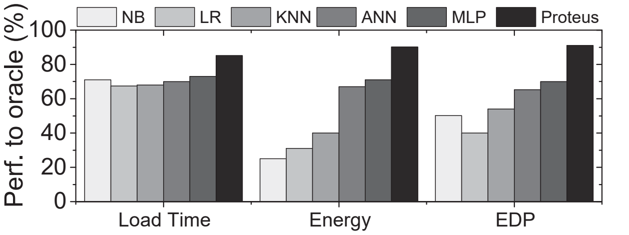


Figure 16: The performance w.r.t oracle achieved by our SVM based approach and other classification techniques.

8 DISCUSSIONS

Naturally there is room for further work and improvements. We discuss a few points here.

Multi-tasking workloads. Our evaluation assumes one single webpage is being downloaded and rendered at a time. Our approach can be extended to a multi-tasking environment of multiple workloads (including web and non-web programs) by focusing on optimizing the web content that is presented on the current screen view. This can be achieved by e.g., moving the background tasks to the little processor cores, or adapting a “selfish" optimization strategy similar to [21, 26, 69]. We expect that knowledge of the web content and network status is still essential for achieving good performance in such an environment.

Changing network conditions. To ensure reproducibility, during evaluation we deterministically replayed the network conditions. In real-world deployment, the network status may change drastically throughout web rendering. To adapt to the changing network conditions, Proteus will need to sample the network status and trigger re-prediction if the network status has changed significantly. Re-prediction should also take into consideration how much web content has been downloaded. Extending Proteus to a dynamic network environment is our future work.

Targeting unseen networks. Proteus is evaluated in seven typical network environments where we train a classification model for each environment and optimization goal (Section 5.2). Proteus can be extended to target unseen network environments. To do so, one can combine classification- and regression-based models. For example, a regression model can be learned to estimate the load time or energy consumption by taking as input the web features and a given processor frequency. In this way, if the current network is significantly different from the one the classification models are tuned for, the regression models can be used to search for an optimal processor configuration. The combination offers a generalisable solution for any network environment.

Different application workloads. Proteus is evaluated on static web contents primarily consist of HTMLs and images. To target dynamic contents such as JavaScript dominated webpages or video streaming, we will need new features to capture the workloads and a mechanism for constant monitoring and frequency adjustment. Given the dynamic nature of the problem, it would be interesting to see whether reinforcement [55] rather than supervised learning is a better way for modeling the problem.

9 RELATED WORK

Our work builds upon the following techniques, while qualitatively differing from each.

Web Browsing Optimization. Numerous techniques have been proposed to optimize web browsing, through e.g. prefetching [64] and caching [46] web contents, scheduling network requests [47], or re-constructing the browser workflow [39, 73] or the TCP protocol [70]. Most of the prior work target homogeneous systems and do not optimize across networking environments. The work presented by Zhu *et al.* [75] and prior work [51] 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.* [14] 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. Proteus avoids the pitfall by developing an approach to automatically learn how to best schedule rendering process. As this work focuses on rendering process mapping, other optimization techniques proposed in [14], such as dynamic buffering, are complementary to our work.

Task Scheduling. There is an extensive body of work on task scheduling on homogeneous and heterogeneous multi-core systems [10, 23, 42, 54, 72]. Most of the prior work in the area use heuristics or analytical models to determine which processor to use to run an application task, by exploiting the code or runtime information of the program. Proteus targets a different domain by using the web workload characteristics to optimize mobile web browsing across networking environments and optimization objectives.

Energy Optimization. Techniques have been proposed to optimize web browsing via application-level optimization, including aggregating data traffic [12, 32, 60] or requests [11, 37], and parallel downloading [8, 31]. Our approach targets a lower level, by exploiting the heterogeneous hardware architecture to perform energy optimization. There is also an intensive body of research on web workload characterization [9, 15, 19]. The insights found from these studies can help us to better extract useful web features.

Predictive Modeling. Machine learning techniques have been employed for various optimization tasks [65], including estimating mobile traffic [52], parallelism mapping [57], code optimization [17, 18, 27, 38, 44, 45, 59, 62, 63, 66–68, 71], task scheduling [21, 25, 26, 50], processor resource allocation [69], model selection [56], etc. Proteus is the first work to use machine learning to predict the optimal processor configuration for mobile web browsing by exploiting the knowledge of the communication network.

10 CONCLUSION

This paper has presented Proteus, 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 with what frequency to use to run the web rendering process and the optimal GPU frequency for running the painting process. 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 two representative heterogeneous mobile multi-cores mobile platforms using the top 1,000 hottest websites. Experimental results show that our approach achieves over 80% of the `oracle` performance, and delivers portable performance by consistently outperforming the state-of-the-art works for load time, energy consumption and `EDP` across evaluation platforms.

ACKNOWLEDGMENTS

We are grateful to our shepherd, Eric Rozner, and the anonymous reviewers for their insightful feedback. This work was supported in part by the National Natural Science Foundation of China under grant agreements 61872294 and 61602501; and the Fundamental Research Funds for the Central Universities under grant agreement GK201803063.

REFERENCES

[1] [n. d.]. big.LITTLE Technology. http://www.arm.com/products/processors/technologies/biglittleprocessing. ([n. d.]).

[2] 2015. Web page Replay. http://www.github.com/chromium/web-page-replay. (2015).

[3] 2016. State of Mobile Networks: UK. https://opensignal.com/reports/. (2016).

[4] 2017. Alexa. http://www.alexa.com/topsites. (2017).

[5] 2017. Intel powerclamp driver. https://www.kernel.org/doc/Documentation/thermal. (2017).

[6] 2018. Chrome. https://www.google.com/chrome/. (2018).

[7] Mohamed M Sabry Aly et al. 2015. Energy-efficient abundant-data computing: The N3XT 1,000 x. *IEEE Computer* (2015).

[8] Behnaz Arzani et al. 2014. Impact of Path Characteristics and Scheduling Policies on MPTCP Performance. In *International Conference on Advanced Information NETWORKING and Applications Workshops*. 743–748.

[9] Alemnew Sheferaw Asrese, Pasi Sarolahti, Magnus Boye, and Jorg Ott. 2016. WePR: A Tool for Automated Web Performance Measurement. In *Globecom Workshops (GC Wkshps), 2016 IEEE*. IEEE, 1–6.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[11] Suzan Bayhan et al. 2017. Improving Cellular Capacity with White Space Offloading. In *WiOpt '17*.

[12] Suzan Bayhan, Gopika Premsankar, Mario Di Francesco, and Jussi Kangasharju. 2016. Mobile Content Offloading in Database-Assisted White Space Networks. In *International Conference on Cognitive Radio Oriented Wireless Networks*. Springer, 129–141.

[13] Joshua Bixby. 2011. The relationship between faster mobile sites and business kpis: Case studies from the mobile frontier. (2011).

[14] Duc Hoang Bui, Yunxin Liu, Hyosu Kim, Insik Shin, and Feng Zhao. 2015. Rethinking energy-performance trade-off in mobile web page loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 14–26.

[15] Yi Cao, Javad Nejati, Muhammad Wajahat, Aruna Balasubramanian, and Anshul Gandhi. 2017. Deconstructing the Energy Consumption of the Mobile Page Load. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 6.

[16] Andre Charland and Brian Leroux. 2011. Mobile application development: web vs. native. *Commun. ACM* 54, 5 (2011), 49–53.

[17] Shizhao Chen et al. 2018. Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures. In *HPCC '18*.

[18] Chris Cummins et al. 2017. End-to-end Deep Learning of Optimization Heuristics. In *PACT '17*.

[19] Salvatore D'Ambrosio et al. 2016. Energy consumption and privacy in mobile Web browsing: Individual issues and connected solutions. *Sustainable Computing: Informatics and Systems* (2016).

[20] George H Dunteman. 1989. *Principal components analysis*. Number 69.

[21] Murali Krishna Emani et al. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*.

[22] Wolfgang Ertel. 1994. On the definition of speedup. In *International Conference on Parallel Architectures and Languages Europe*.

[23] Stijn Eyerman and Lieven Eeckhout. 2010. Probabilistic job symbiosis modeling for SMT processor scheduling. *ACM Sigplan Notices* 45, 3 (2010).

[24] Ricardo Gonzalez et al. 1997. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits* (1997).

[25] Dominik Grewe et al. 2011. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*.

[26] Dominik Grewe et al. 2013. OpenCL task partitioning in the presence of GPU contention. In *LCPC '13*.

[27] Dominik Grewe et al. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO*.

[28] Android Modders Guide. 2017. CPU Governors, Hotplug drivers and GPU governors,. https://androidmodguide.blogspot.com/p/blog-page.html. (2017).

[29] Matthew Halpern et al. 2016. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *HPCA*.

[30] Stephen Hemminger et al. 2005. Network emulation with NetEm. In *Linux conf au*. 18–23.

[31] Mohammad A Hoque, Sasu Tarkoma, and Tuikku Anttila. 2015. Poster: Extremely Parallel Resource Pre-Fetching for Energy Optimized Mobile Web Browsing. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 236–238.

[32] Wenjie Hu and Guohong Cao. 2014. Energy optimization through traffic aggregation in wireless networks. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 916–924.

[33] Connor Imes and Henry Hoffmann. 2016. Bard: A unified framework for managing soft timing and power constraints. In *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*. IEEE, 31–38.

[34] Smart Insights. 2016. Mobile Marketing Statistics compilation. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. (2016).

[35] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. 2007. Supervised machine learning: A review of classification techniques. (2007).

[36] Cody Kwok, Oren Etzioni, and Daniel S Weld. 2001. Scaling question answering to the web. *ACM Transactions on Information Systems (TOIS)* 19, 3 (2001), 242–262.

[37] Ding Li et al. 2016. Automated energy optimization of http requests for mobile applications. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 249–260.

[38] Chen Lindong et al. 2018. Optimizing Sparse Matrix-Vector Multiplications on An ARMv8-based Many-Core Architecture. *International Journal of Parallel Programming* (2018).

[39] Haohui Mai et al. 2012. A case for parallelizing web pages. In *4th USENIX Workshop on Hot Topics in Parallelism*.

[40] Bryan FJ Manly and Jorge A Navarro Alberto. 2016. *Multivariate statistical methods: a primer*. CRC Press.

[41] Leo A Meyerovich and Rastislav Bodik. 2010. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*. ACM, 711–720.

[42] Prasant Mohapatra, ByungJun Ahn, and Jian-Feng Shi. 1996. On-line real-time task scheduling on partitionable multiprocessors. In *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on*. IEEE, 350–357.

[43] Javad Nejati and Aruna Balasubramanian. 2016. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1305–1315.

[44] William F Ogilvie et al. 2014. Fast automatic heuristic construction using active learning. In *LCPC '14*.

[45] William F Ogilvie et al. 2017. Minimizing the cost of iterative compilation with active learning. In *CGO '17*.

[46] Feng Qian et al. 2012. Web caching on smartphones: ideal vs. reality. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 127–140.

[47] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. 2014. Characterizing resource usage for mobile web browsing. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 218–231.

[48] Siddharth Rai and Mainak Chaudhuri. 2017. Improving CPU Performance through Dynamic GPU Access Throttling in CPU-GPU Heterogeneous Processors. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 18–29.

[49] Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. 2018. Two Billion Devices and Counting. *IEEE Micro* 38, 1 (2018), 6–21.

[50] Jie Ren et al. 2017. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM '17*.

[51] Jie Ren, Ling Gao, Hai Wang, and Zheng Wang. [n. d.]. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *IEEE International Conference on Computer Communications (INFOCOM), 2017*.

[52] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 361–374.

[53] Wonik Seo, Daegil Im, Jeongim Choi, and Jaehyuk Huh. 2015. Big or Little: A Study of Mobile Interactive Applications on an Asymmetric Multi-core Platform. In *IEEE International Symposium on Workload Characterization*. 1–11.

[54] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 1.

[55] Richard S. Sutton and Andrew G. Barto. 1998. Reinforcement Learning I: Introduction. (1998).

[56] Ben Taylor et al. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *LCTES*.

[57] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. (2017).

[58] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. 2012. Who killed my battery?: analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 41–50.

[59] Georgios Tournavitis et al. 2009. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *PLDI '09*.

[60] Lorenzo Valerio, F Ben Abdesslemy, A Lindgreny, Raffaele Bruno, Andrea Passarella, and Markus Luoto. 2015. Offloading cellular traffic with opportunistic networks: a feasibility study. In *Ad Hoc Networking Workshop (MED-HOC-NET), 2015 14th Annual Mediterranean*. IEEE, 1–8.

[61] Vlamimir Vapnik. 1998. *Statistical learning theory*. Vol. 1.

[62] Zheng Wang et al. 2014. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM TACO* (2014).

[63] Zheng Wang et al. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO* (2014).

[64] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How far can client-only solutions go for mobile browser speed?. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 31–40.

[65] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* (2018).

[66] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *PPoPP '09*.

[67] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*.

[68] Zheng Wang and Michael FP O'boyle. 2013. Using machine learning to partition streaming programs. *ACM TACO* (2013).

[69] Yuan Wen et al. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *HiPC '14*.

[70] Xiufeng Xie, Xinyu Zhang, and Shilin Zhu. 2017. Accelerating Mobile Web Loading Using Cellular Link Information. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*.

[71] Peng Zhang, , et al. 2018. Auto-tuning Streamed Applications on Intel Xeon Phi. In *IPDPS '18*.

[72] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z Chen. 2002. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference*. ACM, 183–188.

[73] Bo Zhao, Wenjie Hu, Qiang Zheng, and Guohong Cao. 2015. Energy-aware web browsing on smartphones. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2015), 761–774.

[74] Yuhao Zhu et al. 2015. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 137–149.

[75] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and energy-efficient mobile web browsing on big/little systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 13–24.