Improving Smartphone User Experience by Balancing Performance and Energy with Probabilistic QoS Guarantee

Benjamin Gaudette, Carole-Jean Wu, and Sarma Vrudhula
School of Computing, Informatics, & Decision Systems Engineering
Arizona State University
Tempe, AZ 85281
Email: {bgaudett,carole-jean.wu,vrudhula}@asu.edu

ABSTRACT

User satisfaction is pivotal to the success of a mobile application. A recent study has shown that 49% of users would abandon a web-based application if it failed to load within 10 seconds. At the same time, it is imperative to maximize energy efficiency to ensure maximum usage of the limited energy source available to smartphones while maintaining the necessary levels of user satisfaction. An important factor to consider, that has been previously neglected, is variability of execution times of an application, requiring them to be modeled as stochastic quantities. This changes the nature of the objective function and the constraints of the underlying optimization problem. In this paper, we present a new approach to optimal energy control of mobile applications running on modern smartphone devices, focusing on the need to ensure a specified level of user satisfaction. The proposed statistical models address both single and multi-stage applications and are used in the formulation of an optimization problem, the solution to which is a static, lightweight controller that optimizes energy efficiency of mobile applications, subject to constraints on the likelihood that the application execution time meets a given deadline. We demonstrate the proposed models and the corresponding optimization method on three common mobile applications running on a real Qualcomm Snapdragon 8074 mobile chipset. The results show that the proposed statistical estimates of application execution times are within 99.34% of the measured values. Additionally, on the actual Qualcomm Snapdragon 8074 mobile chipset, the proposed control scheme achieves a 29% power savings over commonly-used Linux governors while maintaining an average web page load time of 2 seconds with a likelihood of 90%.

1 Introduction

In recent years, there has been an explosive growth in the use of mobile platforms—especially smartphones—for our everyday computing needs. Applications developed for these devices serve as gateways for many businesses to interact with their clients; however, in a recent survey it has been found that quality of service (QoS) and user satisfaction of web-based applications plays a pivotal role in the financial success of a business [4]. In fact, the study found that 19% of the users would abandon a page requiring more than 5 seconds to load and 49% of the users would abandon the page after 10 seconds. Perhaps worse than this, 79% of the participants stated that if they are dissatisfied with their user experience, they are less likely to shop from the site again. By extrapolating this data, the study concluded that "If an e-commerce site is making $100,000 per day, a 1 second page delay could potentially cost you $2.5 million in lost sales every year." From this perspective, it is imperative to ensure user satisfaction even at the cost of additional power; however, due to finite energy capacity, it is not typically optimal for mobile platforms to sustain high performance states, and as this paper will demonstrate, providing guarantees on QoS requires a different approach that takes into account the distribution of page load times.

Over the past decade, a large body of research has been published on optimizing energy efficiency [10, 13, 16, 18, 23, 24, 38, 43, 44, 47]. However, most, if not all of the work, have assumed that the underlying quantities (e.g., execution times) are deterministic. In [27], Isci et al. considered voltage and frequency (DVFS) as a means to dynamically tune a processor's power consumption based on an application's computation and memory characteristics. Others considered thermal constraints and leakage power when optimizing energy efficiency [20, 22, 28, 31, 33, 35].

While most of the prior works focus on optimizing system energy efficiency for the application domain of chip-multiprocessors (CMPs), a few more recent works have proposed energy efficiency optimization algorithms for user-centric, interactive smartphone applications. Egilmez et al. [14] and Singla et al. [50] proposed temperature-aware energy efficiency optimization while Zhu et al. [52] focused on meeting interactive application QoS. Egilmez et al. [14] specifically aimed at user satisfaction through DVFS-based control knobs to maintain a low and comfortable temperature. Singla et al. [50] developed power and thermal models for a modern mobile platform and proposed a closed loop thermal control to adjust processor frequencies. Zhu et al. [52] proposed eQoS to improve the energy efficiency of web browsers for smartphones while meeting certain QoS constraints.

Although the problems addressed in the existing literature are complex in and of themselves, they have assumed that the execution time of an application is *deterministic*. In reality, execution times vary substantially, depending on the continuously varying states of the system. This is due to factors such as operating system (OS) preemption events, page faults, interrupts, processor pipeline stalls, branch prediction, cache misses, context switches, data input variations, network delays, etc. While considering only the average case may result in higher energy efficiency on average, not accounting for the variations in execution time can lead to situations where user satisfaction is poor. For these reasons, the execution time of an application must be modeled as a stochastic value, whose distribution ideally should depend on the control variables (voltage and frequency) and other parameters that capture the characteristics of the data and

978-1-4673-9211-2/16/$31.00

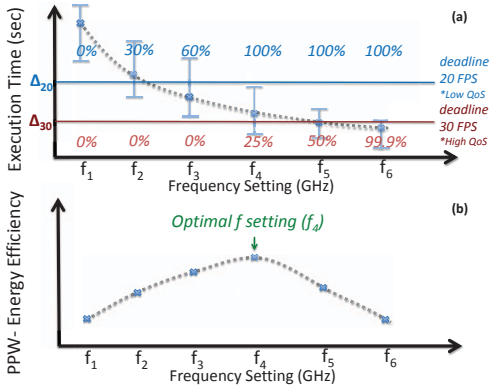


Figure 1: Conceptual plots of mean execution time and PPW of an application running at various clock frequencies, with error bars depicting range of variation, and probability of meeting a given deadline at each frequency.

the computing devices. With this paradigm change, the constraints in the optimization problem can no longer be simply viewed as being satisfied or not being satisfied, but instead must be expressed as a *likelihood* of being satisfied.

To illustrate the importance of considering the likelihood of satisfying constraints, and the potential tradeoffs, consider *video playback*, which is a very common smartphone application. Suppose that a processor allows multiple frequency settings, f_1-f_6. Figure 1(a) shows a *hypothetical* plot of the *average* execution time (dotted curve) of the application (which involves loading, decoding, and displaying a video frame) over multiple invocations, at different frequency settings. As stated earlier, the different invocations at a fixed frequency lead to different execution times due to various factors, such as the frame complexity, and the non-deterministic nature of the system states. The vertical bars at each frequency indicate the observed minimum and maximum execution times. Figure 1(b) shows the average *energy efficiency*, expressed as *performance per watt* (PPW) of the application at each frequency.

The QoS here is a composite measure of how stringent the deadline is for processing a frame, as well as the likelihood of meeting the deadline. Suppose that to satisfy the *high* QoS requires the video to be played at 30 frames-per-second (fps), which translates to a frame execution time 33.3 ms. Let Δ_{30} denote this deadline (see Figure 1(a)). To meet this deadline with a likelihood of 99.9%, will require running the core at frequency f_6. However, at this frequency the average energy efficiency or PPW will be very low. This may result, for instance, in substantially reducing the residual battery charge, raising the possibility of an inoperable phone until the next charging opportunity. However if the user is willing to sacrifice image quality for a longer lasting battery, then the deadline can be increased to 50 ms, which corresponds to 20 fps. Then the phone can be operated at f_4, at which the energy efficiency is maximum, with a near certainty of meeting the larger deadline. Another viewpoint of the tradeoff is that at both f_4 and f_6, there is a very high likelihood that the video will be completed at 20 fps, but at f_4, a substantially larger number of frames will be dropped as compared to frequency f_6, resulting in a low image quality, while ensuring a larger residual battery charge.

Overview: In this paper, we present a new approach to improve mobile user experience by balancing performance and energy with probabilistic guarantee of QoS. The approach is based on modeling the execution time of applications as stochastic quantities. Each application consists of some number of computational segments or basic *units of computation* (UoC), with each UoC being mapped to a core. Variations in execution times are due to different sources of interruptions in the normal flow of computation, as well as intrinsic variations in the complexity of the data being processed. Note that the notion of a UoC varies among applications. We first hypothesize, and then experimentally validate a model (distribution function) of the execution times of UoCs. The model parameters are functions of the control variables (e.g., voltage and frequency), as well as characteristics of the data being processed (e.g., complexity of the web page component being processed).

The model of the total execution time of an application depends on how its constituent UoCs are mapped to the cores. For example, some applications have independent UoCs executing in parallel on separate cores, while in others, the UoCs interact. This leads to different ways of modeling the total execution time for different applications. The proposed statistical models are used in the formulation of an optimization problem, the solution to which is a static, lightweight controller that optimizes energy efficiency of mobile applications, subject to constraints on the *likelihood* that the application execution time meets given deadlines.

The statistical models for single and multi-stage applications and the design of the controller are demonstrated on three commonly used smartphone applications: web browsing, image similarity search, and video playback, and on two different real platforms: an Intel Quad-Core Sandy-Bridge (SNB) processor and a Qualcomm Snapdragon 8074 chipset-based mobile device. The results show that the proposed statistical estimates of application execution times are within 99.34% of the measured values. We then demonstrate an application case study for which these estimates are applied to select the frequency setting when accounting for device energy efficiency and an execution time deadline. Our proposed optimization achieved a 29% power savings over commonly-used Linux governors while maintaining an average web page load time of 2 seconds with a likelihood of 90%—a quality that the Linux governors do not consider. We achieve similar findings for the other mobile applications: image similarity search and video playback.

In summary, this paper proposes an accurate execution time model that provides a quantative measure of the likelihood of meeting a deadline – an overlooked dimension of execution time modeling. To demonstrate the importance of the developed model, we build the model for a modern, high-performance real smartphone device to improve user experience while providing a probabilistic guarantee on QoS. By doing so, on average the QoS is improved by 2.18**x** at a cost of a 25% degradation in power efficiency when compared to the optimal energy efficiency setting.

2 Problem Description

In this section, we introduce some basic terminologies and make a precise statement of the optimization problem. Then, we describe two different scenarios that fit into the general optimization framework, but with different models of execution time.

An application consists of a collection of basic UoCs, each of which is mapped to a core. For instance, a web browser receives an HTML document that consists of a number of URIs (Uniform Resource Index). The URIs are queued, pro-

cessed and ultimately displayed on screen. Therefore, in this instance, a UoC would simply be a URI or a collection of URIs. In a video playback, a UoC refers to a single video frame that undergoes several stages of processing before it is displayed.

The execution time of a UoC on a single core is a random variable X, with a distribution function $F_X(x \mid s) = \text{Prob}(X \leq x \mid s)$, where s represents the core frequency. In general, the distribution function can also include a set of *uncontrollable* parameters that affect the execution time, reflecting the complexity of the data being processed. Examples of such parameters would be number of URIs or the complexity of URIs in a webpage as well as the operating frequencies of each processor core. Note that in general the core voltage setting is determined by the frequency, and hence we consider only the core frequency as the main control variable.

In this work we limit our study to the situation where only one foreground application is being executed at a given time. Currently, this seems to be the most common application use case on smartphones; however, $F_X(x \mid s)$ can be extended to handle multiple applications with enough training to expose the variability caused by the interfering applications. For example in [48] (deterministic) the execution time is parameterized as a function of the memory interference due to concurrently running applications. We could similarly define the execution time pdf parameters to be functions of the memory interference.

A given dataset (e.g., a webpage or a video frame) is a collection of UoCs, $U_1, U_2, \cdots, U_n$, whose corresponding individual execution times are independent random variables $X_1, X_2 \cdots X_n$, with distribution function F_X. In a CMP with n cores, let $Z_n(\boldsymbol{s})$ denote the total execution time of an application, where $\boldsymbol{s}$ denotes the vector of core frequencies. The relation between $Z_n(\boldsymbol{s})$, and the execution time of the UoCs that make up the application depends on the application and how they are mapped onto the cores. For instance, if all its UoCs execute concurrently and independently, then $Z_n(s)$ would be the maximum of the individual execution times, whereas in the case of a single core $Z_n(\boldsymbol{s})$ would be the sum of the individual execution times.

The objective function to be maximized is energy efficiency, also referred to as *performance-per-watt*, denoted by $\boldsymbol{E}(PPW(\boldsymbol{s}))$, where $\boldsymbol{E}()$ denotes the expected value. PPW is the ratio of the performance to power, and performance is simply the reciprocal of the execution time. The measure of performance depends on the specific application, but is in general, a function of all the core frequencies $\boldsymbol{s}$. In the case of a video playback, $PPW(\boldsymbol{s})$ would denote the average number of fps per Watt, and in the case of a web browser, it would be the number of webpages displayed per second per Watt. Hence, PPW denotes the number of items processed per joule of energy. Let Δ denote a deadline and $Q \in (0,1)$ denote a lower bound on the likelihood of satisfying the given deadline. Then, with these notations, the optimization problem can be stated as follows:

$$\max_{\mathbf{s}} \quad \boldsymbol{E}(PPW(\mathbf{s})) \tag{1}$$

$$\text{s.t.} \quad F_{Z_n(s)}(\Delta|\mathbf{s}) = \text{Prob}(Z_n(\boldsymbol{s}) \leq \Delta) \geq Q. \tag{2}$$

A naive approach to solving the above optimization problem would be to experimentally evaluate the application program at all combinations of per-core frequencies. Given the large number of available frequency settings in modern processors and the increasing number of cores in the application processor, a brute force evaluation for identifying the optimal frequency combination for all cores is infeasible. For example, there are fourteen available frequency settings per core in the Qualcomm Snapdragon chipset (ranging from 300MHz to 2.15GHz). For an Octa-core application processor, it requires 14^8 experiments to determine the frequency combination that maximizes system energy efficiency for any application. For this reason, the more sophisticated modeling mechanisms presented herein, are needed. The proposed approach addresses two distinct scenarios, one where all the cores are operating concurrently and independently, and the other where there are interactions among the cores. These two scenarios require a different approach to modeling the execution times.

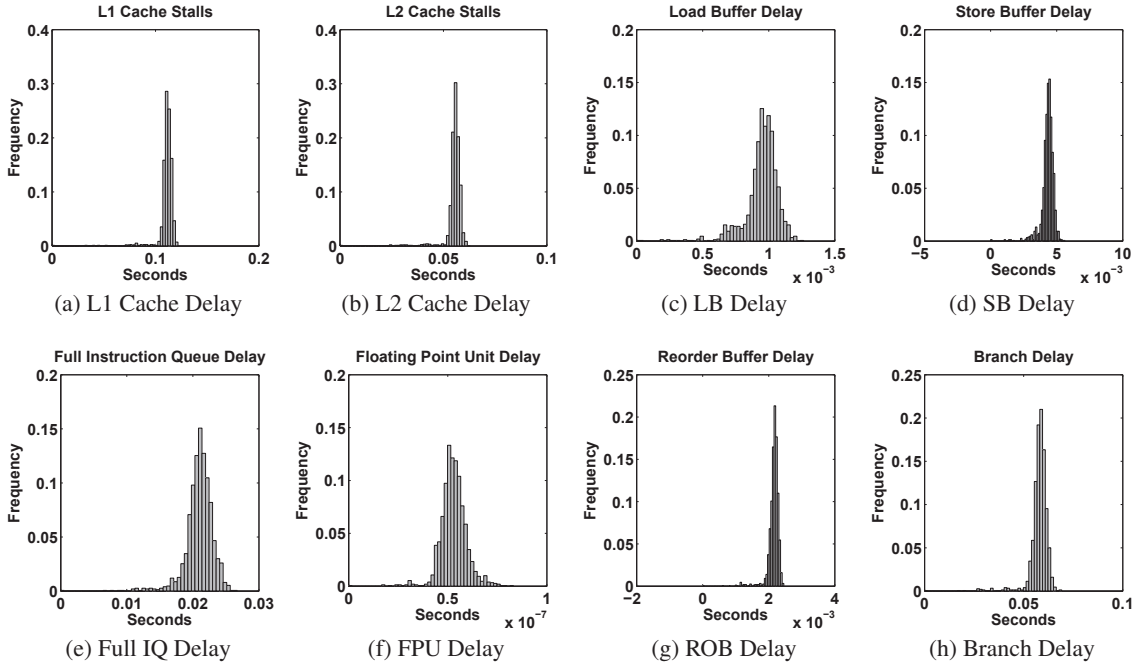
While this work only examined core frequencies due to the lack of control over other SoC components (GPU, DSP, etc.), incorporating accelerators will require no changes to the models presented. It is only necessary to monitor the runtime of the code segment(s) offloaded and develop probability distributions as a function of their controls.

3 Execution Time Models

3.1 Computations on a Single Core

We start this section with some evidence that supports the need to model execution times as random variables. As stated earlier, during the course of execution, flow of computation will be subjected to numerous interruptions, which contribute to its execution time. To better understand the sources of interruptions, we carried out an experiment on the Intel SNB processor[1] in which the architectural sources of interrupts were monitored using the available performance counters on the processor [25] (the complete methodological detail is given in Section 5). It is important to note, that the severaity of each source of delay can vary greatly. In our work we focus on the performance of web page rendering (web pages are loaded offline) and thus network delay is not included. That said, network delay is not necessarily the dominating factor on mobile phones. A study from [29] showed that under a 2Mbps network connection halving the CPU frequency results in a 50%-100% increase in the page load times, thus the bottleneck becomes the client CPU. To avoid the correlation between cores, we disabled all but one application processor core. The Firefox web browser was executed, along with the eleven most visited web pages from the BBench suite [19], including Amazon, BBC, CNN, Craiglist, eBay, ESPN, Google, MSN, Slashdot, Twitter, and YouTube. The various sources of delays were monitored during the course of the eleven webpage loads. This was repeated 1,500 times, and the execution times and the associated delays incurred in the processor and the memory subsystem were recorded. Table 1 shows the statistical information gathered from the experiment, and Figures 2(a-h) show the histograms of the durations of each of the monitored sources of interruptions. The data clearly demonstrates a substantial

[1]The Intel SNB processor was selected over other platforms due to the availability of advanced performance counters to measure the numerous sources of delays. While the mobile platforms may have some performance counters, the available information is not sufficiently detailed as that given by the Intel SNB processor. Although the parameters of the distributions may change from platform to platform, the underlying distributions will not.



(a) L1 Cache Delay (b) L2 Cache Delay (c) LB Delay (d) SB Delay

(e) Full IQ Delay (f) FPU Delay (g) ROB Delay (h) Branch Delay

Figure 2: Distributions of the non-deterministic portions of sources of delay (Minimum delays subtracted).

Table 1: Summary of Sources of Delays

	Mean	Std. Deviation	Min
Execution Time (s)	0.5532	0.2117	0.2862
Source of delay	**Mean**	**Std. Deviation**	**Min**
L2 Cache Stalls (s)	0.0828	0.0048	0.0276
L1 Cache Stalls (s)	0.1651	0.0095	0.0546
LB Stalls (s)	0.0012	0.0001	0.0003
SB Stalls (s)	0.0063	0.0005	0.0020
Full IQ Stalls (s)	0.0326	0.0021	0.0116
FPU Stalls (s)	8.80E-8	6.71E-9	3.51E-8
ROB Stalls (s)	0.0008	0.0002	0.0007
Branch Stalls (s)	0.0858	0.0049	0.0280

variations in the durations of different types of interruptions. In particular, the standard deviation of the total execution time (0.2117) is about 38% of the mean.

The interruptions to the execution flow can be viewed as intervals of idle periods, whose endpoints are random points in time, and whose lengths are random variables. Thus the total execution time may be viewed as the sum of some minimum execution time (not random, but unknown) and the total duration of all the interruptions. A model for sum of random interval lengths often used in the literature is the Gamma distribution [6]. Figure 3 shows the histogram of the total execution of the web browser compared to a 3-parameter Gamma distribution. It shows that the Gamma distribution provides an adequate model to explain the variations in the total execution time for this application on a single core. Note that the dependence of the probability distribution function (pdf) on the core frequency will be modeled by relating it to the parameters of the Gamma distribution.

The pdf of the Gamma is given by

$$f_{X(s)}(x) = \text{Gamma}(x; K(s), \theta(s), \lambda(s)) = \frac{1}{\theta(s)\Gamma(K(s))}\left(\frac{x-\lambda(s)}{\theta(s)}\right)^{K(s)-1}, \quad (3)$$

s is the core frequency, $K(s)$ is the shape parameter, $\theta(s)$ is the scale parameter, and $\lambda(s)$ is the left endpoint or minimum value of $X(s)$. The mean $\mu(s)$ and variance $\sigma^2(s)$ of

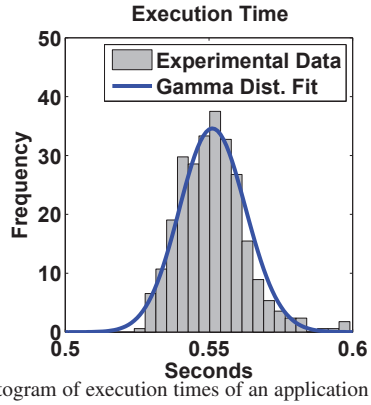


Figure 3: Histogram of execution times of an application running on a single core, compared to a pdf of a 3-parameter Gamma pdf.

the distribution are given by: $\mu(s) = K(s)\theta(s)$ and $\sigma^2(s) = K(s)\theta^2(s)$.

It is important to note that the parameters $K(s)$, $\theta(s)$, and $\lambda(s)$ are modeled as functions of the core frequencies s. This provides a simple and effective means to relate the distribution function of the execution times to the core frequencies. In addition, it is also possible[2] to make K and θ depend on application specific parameters. For instance, for web browsing we could include quantifiable characteristics of the web page such as the number of URIs and the types of objects in the URIs that affect the execution time. In this way, the execution time of a browser in processing a web page can be made sensitive to the complexity of the webpage.

3.2 Parallel Computations, Independent Cores

We now describe the execution time model of an application in which the UoCs are executed in parallel, and independently on multiple cores. Although the description is based on a specific multi-threaded web browser, it is easily made applicable to other browsers and data parallel applications.

[2]Not considered in this article

To load a web page, a browser must process a collection of URI elements. The web browser analyzes the collection of URI elements in a page and determines a subset of elements to be serviced by a secondary instance of the browser, assuming it deems the subset of elements to be "sufficiently complex". The servicing of this subset of URIs mapped to an application processor core denotes a unique UoC with its own execution time distribution. This process can be repeated until all URI elements are assigned a UoC. In practice the web pages of most web sites are partitioned to 1 or 3 UoCs.

With this view, the total time to load a page will be the latest completion time among all the UoCs. Thus if a page results in N UoCs, each mapped to a core, and the execution time of UoC n is a random variable X_n, whose distribution function at a given core frequency s_n is given by (3), the total execution time will be $Z_N(\boldsymbol{s}) = max_{n=1}^{N} X_n(s_n)$, where s_n is the core frequency at which UoC n is processed. The distribution function of $Z_N(\boldsymbol{s})$ is given by

$$F_{Z(\boldsymbol{s})}(z) = \prod_{n=1}^{N} F_{X_n(s_n)}(z). \tag{4}$$

3.3 Parallel Computations, Interacting Cores

The previous sections described how the execution time of independent UoCs can be accurately modeled as simple functions of one or more independent random variables. While this is perfectly suitable to model many applications, there are many others in which UoCs possess a pipelined data flow structure such as *image similarity search* and *video playback*. In this section, we describe how to model the execution time of a collection of UoCs constituting an application that are running in parallel on a network of interacting cores, where the network is a *cascade* of stages.

The naive approach is to model the total execution time of the application as the sum of execution times of the stages. This leads to very pessimistic results (demonstrated in Section 5, see Figure 6) due to the fact that it doesn't accurately model the temporal dependencies among stages. A more appropriate model is the mathematical framework of *Network Calculus* [15] that can be used to describe the computation as flows of data through a network. With nodes exhibiting statistical variations in their execution times, the flows, which are represented by functions of time, are now *stochastic processes* (as opposed to simple random variables). Here, we describe how the combination of network calculus and probability calculus can be used to predict the execution time of parallel interacting tasks.

In its original formulation, Network Calculus (NC) was aimed at modeling deterministic queuing systems for computer networks and is analogous to system theory used in circuit analysis and other domains. It captures the intricacies associated with buffering systems and pipelining in networked systems. The fundamental distinguishing feature of NC is the use of *min-plus* algebra as the basis for its calculations. As such, data flows are modeled as *non-decreasing* functions of time.

NC works on the notion of an observable node or *system*, $\mathscr{S}$, and derives properties (e.g., backlog, delay, etc.) based on the flow of data entering, exiting, and being serviced by the node. $\mathscr{S}$ in the present case is a cascade of one or more nodes. The flows are represented by the non-decreasing functions $I(t)$, $O(t)$, and $S(t)$ which are defined

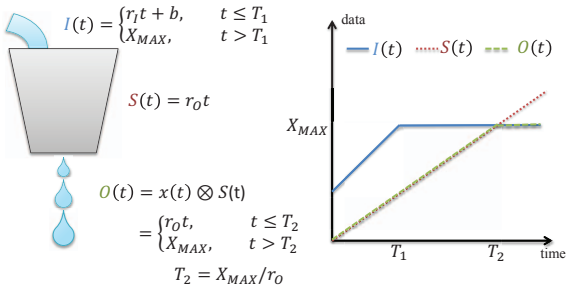


Figure 4: A classical network calculus example, the leaky bucket. The leaky bucket drains water via a hole in the bottom at a constant rate, r_O (i.e. the service characteristics of the system). The system receives an input flow of $r_I t + b$ until time T at which time no more flow is received. The output flow can therefore be calculated via min-plus convolution of the service curve and the input curve.

as follows.

- $I(t)$ – the total amount of data which has entered $\mathscr{S}$ during the interval $[0,t)$.
- $O(t)$ – the total amount of data which has exited $\mathscr{S}$ during the interval $[0,t)$.
- $S(t)$ – the *service curve* of system $\mathscr{S}$. This curve represents a lower bound on the total amount of data which could be serviced during the interval $[0,t)$.

The *goal* of this section is to provide a method to compute the pdf of $O(t)$, the throughput of a system, denoted by $f_{O(t)}(x)$. In the case of a pipelined computation flow, this would be the pdf of the output curve of the last stage. The output flow of a system with a given service curve and input flow is given by

$$O(t) = I(t) \otimes S(t), \tag{5}$$

where $\otimes$ represents the min-plus convolution operation, defined by

$$O(t) = \inf_{0 \le \tau \le t} (I(s) + S(t-\tau)). \tag{6}$$

Figure 4 illustrates a classical example of this operation.

NC provides an algebraic representation of concatenation of simple systems to form a complex networked system. As in system theory, this greatly simplifies the calculation of flows between multiple connected systems. Consider the situation in which two systems $\mathscr{S}_1$ and $\mathscr{S}_2$ possess service curves $S_1(t)$ and $S_2(t)$ respectively. Additionally, $\mathscr{S}_1$ has input flow $I_1(t)$ and output flow $O_1(t)$ while $\mathscr{S}_2$ has input flow $I_2(t)$ and output flow $O_2(t)$. The outflow of the cascade of $\mathscr{S}_1$ and $\mathscr{S}_2$ is given by

$$O_2(t) = I_1(t) \otimes S_1(t) \otimes S_2(t). \tag{7}$$

3.3.1 Distribution of the Service Process S(t)

In deterministic systems, a service process is a monotonic function which describes the potential cumulative output of the system given that the system is never starved for input. In other words, it is akin to the impulse response function in system theory. When extending into systems with non-determinism, a single service curve no longer represents the system. More precisely, it is a stochastic process, which is an infinite, non-denumerable set of service curves, as illustrated in Figure 5. At each fixed time t, the sampled space of outcomes are the points on all the curves at time t, representing the random variable $S(t)$. The objective now is to compute the distribution of the output variable $O(t)$ given by Equation 6, in terms of the input random variables $I(t)$ and

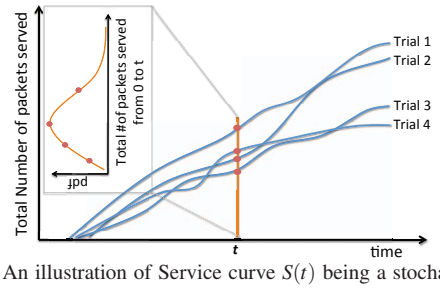


Figure 5: An illustration of Service curve $S(t)$ being a stochastic process. Each trial represents the algorithm being executed under the same input and system controls; however due to random system variations, the service times can vary, thus the observed output process can vary from trial to trial. At each t, the set of points on all the possible (infinite) curves represents the sample space of the random variable $S(t)$.

the service random variable $S(t)$. We may view this as pdfs being propagated through the network to finally compute the pdf of the number of UoCs processed at the system output for each t.

Let $S(t \mid s)$ denote the service curve of some stage, which represents the number of UoCs processed in the interval $[0,t)$. We assume that a stage is synonymous with a core, whose frequency is s. Let $T_n(s)$ denote the time to process n UoCs by a stage. $T_n(s)$ is equal to the sum of the times to process each UoC. Then using the previous notation, let $X(s)$ be the random variable that represents the time to process one UoC by the given stage. The pdf of $X(s)$ is given by Equation (3). The the pdf of $T_n(s)$ is then given by

$$f_{T_n(s)}(t \mid n,s) = \underbrace{f_{X(s)} * f_{X(s)} * \ldots * f_{X(s)}}_{n\text{-fold convolution}}. \quad (8)$$

3.3.2 *Distribution of the Output Process O(t)*

Consider the situation of a single UoC running on a core at frequency s, with an input flow, $I(t)$, service curve, $S(t \mid s)$, and output flow, $O(t \mid s)$. The computation of $O(t \mid s)$ given by Equation 6 is approximated by partitioning the time interval into a discrete set of time points $(0,t_1,t_2,\ldots t_r = t)$.

$$\begin{aligned} O(t \mid s) &= \min_{\tau \in (0,t_1,t_2,\ldots t_r)} I(\tau) + S(t-\tau \mid s) \quad (9)\\ &= \min\{I(0)+S(t), I(t_1)+S(t-t_1 \mid s),\\ &\quad \cdots, I(t_{i-1})+S(t-t_{i-1} \mid s)\} \end{aligned}$$

The density of each sum within the min, i.e. $I(t_k)+S(t-t_k \mid s)$, is computed by the convolution of the individual densities. The density of the minimum of some n random variables, is approximated by iteratively computing the density of the minimum of pairs of random variables, which is given by

$$\begin{aligned} f_{min(U,V)}(x) &= f_U(x) + f_V(x)\\ &\quad - f_U(x)F_V(x) - F_U(x)f_V(x). \quad (10) \end{aligned}$$

For an N stage pipeline, we are interested in the distribution of the output process of the Nth stage, namely $O_N(t \mid \boldsymbol{s})$. Equation 9 is expanded, with the output process of the stage i serving as the input process for stage $i+1$. Then the resulting expression will be the minimum among a potentially large number of service curves, along with the distribution of the initial $I(t)$, which is assumed to be known. The distribution function of such a minimum has to be computed numerically, using the approximation given in Equation 10.

In summary, estimation of execution times is done by numerically computing its distribution using Equation 3 or computing $f_{O_N(t)}(x \mid \boldsymbol{s})$ using the approximation given in Equation 10, depending on whether it is a collection of independent UoCs running on cores or a pipeline computation.

4 Power Model

A vital part of the optimization process is accurate power models. In this work power is represented as the sum of the dynamic power P_{dyn}, the leakage power P_{lkg} and static system power P_{stc}. While P_{stc} represents a constant offset, the dynamic power varies linearly with the clock frequency, as a circuit is operated only when the clock is high, while dynamic power varies quadratically with the voltage, as power of a transistor is the product of transistor current and voltage, and current of a transistor is also a function of voltage. The components of the dynamic power vector are expressed as

$$P_{dyn,c,b}(t) = P^{max}_{dyn,c,b}(t)s_c(t)v_c^2(t), \forall c,b,t \quad (11)$$

where $P^{max}_{dyn,c,b}$ is the dynamic power dissipated by block b of core c when the core is at the maximum speed and voltage. $P^{max}_{dyn,c,b}$ is obtained by profiling the time-varying power consumption of the task to be run on core c.

The leakage power is known to have exponential dependence on the die temperature and supply voltage. The exact equation is hard to derive analytically. Hence it is usually derived based on data fitting of the simulated power values for various components like adders, multipliers, memories, etc. An example empirical equation for leakage power in 65 nm is given [37].

$$\begin{aligned} P_{lkg,c,b}(t) = k^1_{c,b}v_c(t)T^2_{c,b}(t)e^{\frac{\alpha_{c,b}v_c(t)+\beta_{c,b}}{T_{c,b}(t)}} +\\ k^2_{c,b}e^{(\gamma_{c,b}v_c(t)+\delta_{c,b})}, \forall c,b,t. \quad (12) \end{aligned}$$

$k^1_{c,b}$, $k^2_{c,b}$, $\alpha_{c,b}$, $\beta_{c,b}$, $\gamma_{c,b}$, and $\delta_{c,b}$ are parameters that depend on circuit topology, size, technology and design. It should be noted that (12) can be simplified to a piece-wise linear approximation as shown in [23]. Thus we use the following model to estimate leakage power for each core/block on the processor.

$$P_{lkg,c,b}(t) = P_{lkg0,c,b} + G^T_{c,b}T_{c,b}(t) + k^v_{c,b}v_c(t), \forall c,b,t. \quad (13)$$

where $G^T_{c,b}$ and $k^v_{c,b}$ represent the temperature and the voltage coefficients. $P_{lkg0,c,b}$ represents the leakage power for block b in core c corresponding to $T_{c,b} = 0$ and $v_c = 0$ (i.e. the ambient temperature and minimum voltage).

5 Experimental Setup

In this section, we describe the setup of the experiments and the validation results for the execution time and power models, evaluated on real platforms.

5.1 Real-Device Experimental Platform

The experiments were conducted on two different platforms: the Intel Quad-Core SNB processor and a DragonBoard development board based on the Qualcomm Snapdragon 8074 chipset. The Intel SNB processor includes four cores, each of which has a private L1 instruction cache (32KB), a private L1 data cache (32KB), and a private unified L2 cache (256KB). All four cores share the last-level cache of 6144KB. The frequency settings available in the Intel SNB processor range from 0.8GHz to 2.1GHz in steps of 100MHz. Since this processor is not a mobile chipset, it is only used in the event when detailed performance counters are needed

(see Section 3). On the other hand, the Qualcomm Snapdragon 8074 chipset, which is in the US versions of Samsung Galaxy S5 smartphones and many other modern smartphone devices, is used for all results presented in the evaluation section (Section 6). It has four 2.15GHz cores with independent frequency control. Each core hosts a private L0-cache (4KB) and L1-cache (16KB). All cores share a 2MB L2 cache. Since the Dragonboard offers no onboard power sensors, an NI DAQ unit was used with a 1MHz sampling rate for current and voltage measurements. The readings were taken after AC to DC conversion.

Table 2: Parameters for the DragonBoard experimental platform.

	Cortex-A15
Architecture	ARM v7, Krait
Frequency	0.3-2.15GHz
L0 Cache Size	4KB I & 4KB D
L1 Cache Size	16KB I & 16KB D
L2 Cache Size	2MB

The experimental platforms run a rooted Android 4.4 KitKat OS. The applications of interest *web browsing* [2], *image similarity search* [8], *video playback* [3], were written in C and cross-compiled on the host machine with the ARM-Android NDK toolchains [1]. The binary is pushed to the device and is launched from the host machine via the adb terminal.

5.2 Benchmark Applications

Three applications common to the mobile domain were evaluated. The first is *web browsing*. Following [53], a web page is viewed as a collection of elements (or URIs) which are then serviced by the browser's threads. Each thread is treated as an independent server and acts in parallel with other threads. The specific web browser was Firefox. Minor modifications were made to the source code in order to observe the service time of each web page and each URI, to compute the empirical distributions. The BBench 2.0 benchmark [19] was used as it contains a collection of the eleven most widely viewed webpages. It should be noted that this benchmark is limited to offline browsing due to the limitations of implementating timing analysis code into the web pages. Although not explored in this work, it is possible to capture the effects of network delay by either (1) decreasing the value of δ by the expected value of the network delay or (2) model the network as UoC cascading into the remainder of the stages. Effectively this creates an arrival process for the web browser. The second approach is explored in more detail in [34]. That said, network delay is not necessarily the dominating factor on mobile phones.

The second application was *image similarity search*, which is an image similarity ranking algorithm from the PARSEC benchmark suite [8]. The algorithm is used for content-based similarity search of feature-rich data, such as images or videos, and is an important building block for many image recognition and augmented reality Apps running on modern smartphones or Google Glass-like devices. The algorithm consists of six stages of processing. Given an image, *image similarity search* searches a database of images to find the closest match. After the image is loaded, it is first fragmented into segments based on its contents. Then the feature extraction stage assigns each fragment a numerical feature vector. Finally this feature vector is compared against a central database and the most similar results are produced as the output. Minimal source code modifications were made to control binding of specific threads to cores in order to reduce variability in testing and to ensure that the proper cluster of cores were being tested. The load/input, feature extraction, and output stages were bound to the same core due to their relatively light computation, whereas the segmentation, indexing, and ranking stages were each mapped to an individual core. The application was run with the *sim-large* input set.

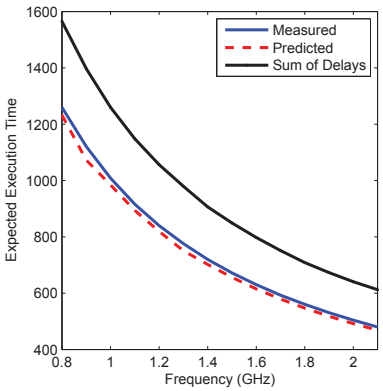


Figure 6: Predicted versus measured execution time of the image similarity search application on Intel SNB quad core processor.

The third application, was a *video playback*, implemented as a data parallel streaming application. We used the open source VLC video player after making minor modifications to the source code in order to observe the per-frame service time and to construct the empirical distributions. The workload was selected from MobileBench [42].[3] Hardware acceleration was disabled so that all video decoding was done by the application processor. The choice was made for two reasons: (1) reaching 30 fps with GPU assisted acceleration would be trivial, and (2) the platforms provide a greater level of control over the application processor, including per-core DVFS in the case of the Snapdragon chipset.

5.3 Validation of Execution Time and Power Models

We demonstrate here the appropriateness and value of NC model for modeling the execution time of a pipeline that involves interacting cores. Figure 6 plots the expected execution time for *image similarity search* that was obtained using the proposed statistical models in combination with NC, and the actual measured values on the Intel SNB processor. It also shows the results of the naive approach, which is simply the sum of the execution times of each stage. The experiment demonstrates the value and accuracy of the proposed approach. The naive approach is a significant overestimate of the actual execution time, which would result in a suboptimal solution of core frequencies when attempting to maximize performance or energy efficiency. It is due to the fact that a simple summation does not account for the overlapping of computations involving different data sets. On average, difference between the model predicted execution times and the observed values was less than 0.66%. The average error using the naive approach was 26%.

The parameters of the model of power described in Section 4, were estimated using data gathered from synthetic

[3]Although all input files were evaluated, we reports results for `big_buck_bunny_720p_stereo.avi` as these are the few formats that require significant amount of rendering time (i.e. obtaining 60 FPS was not trivially done at the lowest speed).

benchmarks aimed to stress the various portions of the microarchitectural and memory components. 2000 samples of power data were collected while running *image similarity search*. The difference between the model predicted power and measured values was 1.72%.

5.4 Optimal Selection of Core Frequencies

The key component of the model creation is that only the independent tasks are characterized. This avoids a full experimental exploration of the state space such that each core only needs to be evaluated at each frequency (rather than evaluating the system at every possible frequency-core combination). This was used to construct the *individual densities* $f_{X(s)}(x;K(s),\theta(s))$ (Equation 3) for each of the service threads.

With each $f_{X(s)}(x;K(s),\theta(s))$, the statistical model was used to compute the joint distribution of multiple random variables, where each corresponds to a core at an individual frequency. For the *web browser* and *video playback*, the joint distribution was computed using Equation 4. For *image similarity search* the joint distributions were computed using the NC approach described in Sections 3.3.1 and 3.3.2.

To find the appropriate optimal frequency combination to solve Equations 1–2 an exhaustive enumeration of the operating frequencies was performed. The problem can then be resolved for various levels of QoS and stored in a table. This table is then used at run time to select the optimal frequency. While this a fairly heavy overhead (in our case 4 cores, 14 frequencies), it is a one-time static analysis which can bring significant improvements in energy efficiency (see Section 6).

In terms of practicality, our current method is very similar to characterizing a standard cell for a VLSI library. For minor application updates or platform differences, the old models could be used-the exact loss of efficiency however is difficult to quantify. Additionally, much like a standard cell needs to be re-characterized for different process technologies, our model needs to be constructed on a per-device basis or given significant application changes. While this is a limitation we believe that the resulting increase in energy efficiency makes the process worthwhile. That said, one of our current efforts is to adapt the methodology to different platforms.

6 Case Study of Balancing Performance and Energy with Probabilistic Guarantee of QoS

In this section we compare the results of using the proposed methodology to maximize the PPW with several standard schedulers available on Android OS: *powersaver*, *performance*, and *interactive* [41]. As stated earlier, the QoS $= (\Delta, Q)$, is a composite metric that includes both the deadline Δ and the likelihood Q (see Equation 2). For the following experiments we set $Q = 0.9$. That is, the selected frequencies should be such that the probability of meeting the deadline Δ is at least 0.9. For this set of experiments, since Q is fixed, QoS is effectively the same as the deadline Δ. For each application two levels of QoS where selected to represent a high and low state. Additionally, we provide an unconstrained solution as well—Maximum Energy Efficiency (Max Energy Eff.). Together these three points can represent different tradeoffs between energy efficiency and user satisfaction, as shown in Figure 7 and Table 3.

For *web browsing* and *image similarity search*, the selected levels of QoS (high, low, and unconstrainted) resulted in greater energy efficiency than the linux governors. The frequency governor, *interactive*, tends to over-react to processor utilization and attempts to set the processor to the highest available frequency (i.e. assumes it is most energy efficient to finish the job as quickly as possible and then slow/disable the processor). As shown in Table 3, the proposed optimization method determines independent frequency levels for each application processor core. This is due to imbalances in the workloads between the cores. For example in the case of *web browsing* the first core receives a significant portion of the total number of URIs which constitute that core's UoC. Therefore its frequency is set to be the highest of 1.72GHz. In contrast, the other cores receive very few URIs which are typically not too computationally demanding. Thus the frequency of these latter cores are reduced. Because there was never a case when four UoCs were simultaneously being executed, the last application processor core's frequency was set to zero, i.e., automatically disabled, to minimize power consumption. This agrees with the observation made in a recent mobile device utilization study [17]—the processor utilization of web browsing has peaks and valleys between one to three cores of a four-core application processor with an average utilization to be below two cores. Similarly, the *image similarity search* application we see that the optimization determines frequencies based on the imbalances in the application pipeline. Specifically we can see that the most computationally demanding stage is mapped to the fourth core and thus always requires a substantially larger frequency than the first three cores.

An interesting situation occurred in the case of video playback. As seen in Figure 7 and Table 3, the lowest frequency setting (i.e. Powersave Governor) results in the optimal PPW point for the system. The unconstrained optimal point of operation occurs at the same frequency setting. For this reason it is very tempting to choose a very slow speed to yield the highest energy efficiency; however, the data shows that the QoS would dramatically drop (given a Q value of 0.9). Thus the video would be playing at 10 fps or lower with a likelihood of at least 90%. It is quite reasonable to assume this to be intolerable for a large user base. This issue highlights the importance of adding constraint (2) to the PPW optimization problem.

There is a strong non-linear relation between PPW and QoS ($= \Delta$). This is more prominent in *video playback*. Table 3 shows that increasing the required fps from unconstrained to 20 fps only degrades PPW by 2.9%, whereas increasing it from 20 to 30 fps degrades PPW by 51.3%. This must be considered when balancing user experience and energy savings.

We reiterate the key points found in our analysis:

- The unconstrained (no consideration to user satisfaction) optimization problem results in the highest possible energy efficiency.
- Including a QoS constraint reduces the possible optimal energy efficiency. Alternatively, reducing Δ (i.e. requiring the application to finish earlier) or increasing Q (i.e. requiring a larger spread of execution times to achieve the chosen deadline) requires a greater amount of power thus reducing energy efficiency.

Next, we examine the accuracy or tightness of the likelihood constraint given in Equation 2, by plotting the observed

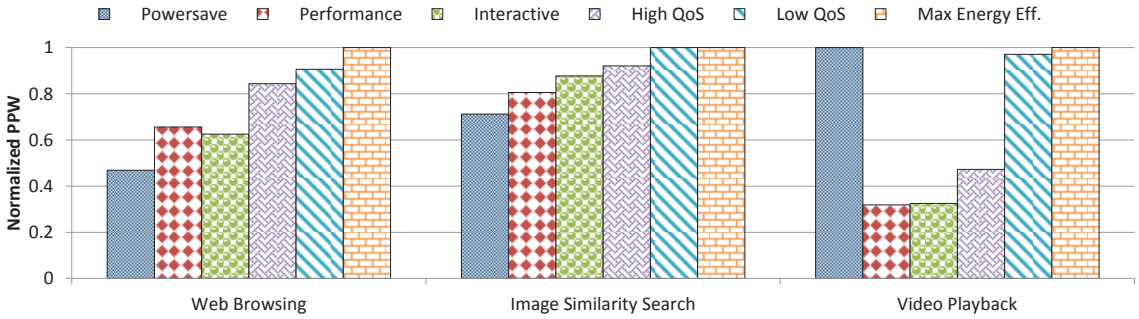


Figure 7: PPW scores of all frequency governors. Higher is better.

Table 3: Performance of several frequency governors on various mobile applications.

Governor	Req. QoS Lower is better	Web Browsing Freq. Vector	Obs. QoS Lower is better	Power	PPW
Powersave	—	300MHz, 300MHz, 300MHz, 300MHz	7.55 Sec	0.90 W	0.15
Performance	—	2.15GHz, 2.15GHz, 2.15GHz, 2.15GHz	0.97 Sec	4.99 W	0.21
Interactive	—	varies over time	1.52 Sec	3.56 W	0.20
High QoS	2.0 Sec	1.72GHz, 1.57GHz, 1.57GHz, off	1.36 Sec	2.76 W	0.27
Low QoS	4.0 Sec	1.26GHz, 960MHz, 960MHz, off	2.12 Sec	1.61 W	0.29
Max Energy Eff.	—	960MHz, off, off, off	4.24 Sec	0.75 W	0.32
Governor	**Req. QoS Higher is better**	**Image Similarity Search Freq. Vector**	**Obs. QoS Higher is better**	**Power**	**PPW**
Powersave	—	300MHz, 300MHz, 300MHz, 300MHz	0.84 Images/Second	0.85 W	0.99
Performance	—	2.15GHz, 2.15GHz, 2.15GHz, 2.15GHz	6.06 Images/Second	5.42 W	1.12
Interactive	—	varies over time	6.05 Images/Second	4.96 W	1.22
High QoS	4 Images/Second	652MHz, 422MHz, 422MHz, 1.49GHz	4.22 Images/Second	3.32 W	1.28
Low QoS	2 Images/Second	422MHz, 422MHz, 300Mhz, 960MHz	2.05 Images/Second	1.94 W	1.39
Max Energy Eff.	—	422MHz, 422MHz, 300MHz, 960MHz	2.05 Images/Second	1.94 W	1.39
Governor	**Req. QoS Higher is better**	**Video Playback Freq. Vector**	**Obs. QoS Higher is better**	**Power**	**PPW**
Powersave	—	300MHz, 300MHz, 300MHz, 300MHz	22.4 FPS	0.95 W	23.9
Performance	—	2.15GHz, 2.15GHz, 2.15GHz, 2.15GHz	39.2 FPS	5.15 W	7.61
Interactive	—	varies over time	39.1 FPS	5.03 W	7.77
High QoS	30 FPS	1.72GHz, 1.72GHz, 1.72GHz, 1.72GHz	31.7 FPS	2.81 W	11.3
Low QoS	20 FPS	652MHz, 652MHz, 652MHz, 652MHz	23.3 FPS	1.00 W	23.2
Max Energy Eff.	—	300MHz, 300MHz, 300MHz, 300MHz	22.4 FPS	0.95 W	23.9

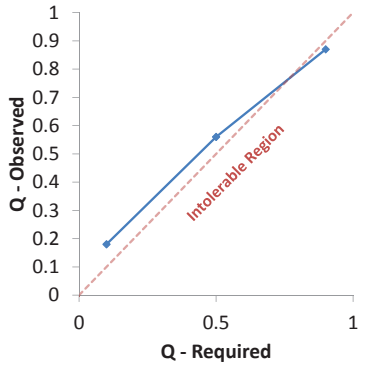


Figure 8: Observed Q versus specified Q (lower bound on likelihood), showing tightness of constraint in Equation 2.

or actual value of Q versus the required or specified value of Q. This is shown in Figure 8, using *web browsing* as an example. The Firefox web browser was repeatedly executed 100 times with Q set to 0.1, 0.5, and 0.9, and $\Delta = 2.0s$. Then the number of page loads that exceeded Δ (2 second load time) was recorded. Since the specified value of Q is a lower bound on the likelihood, and the observed Q is simply a sample estimate of the likelihood, all the points in the plot should be at or above the $y = x$ line. The observed results are very consistent with the expected, and show that the constraint on the likelihood does indeed restrict the set of optimal solutions.

We now demonstrate the potential tradeoff analysis (hypothetical plot shown in Figure 1) that is made possible with the proposed approach. Specifically we explore *web browsing* in greater detail to better demonstrate the benefits of modeling execution time as a non-deterministic value. Figure 9(b) presents the energy efficiency of *web browser* at the corresponding frequency settings. The green circles in Figure 9(a) represent the average execution time needed to load the web page. The range for each point represents the statistical distribution of the execution time for *web browsing* in the various frequency settings.

In addition to the expected load times of the *web browser*, the proposed framework also provides a *quantitative measure* for the likelihood of performance guarantee. This is shown in Figure 9(c) where a deadline on the web page load times is set to 2 seconds. The likelihood of reaching this deadline at a frequency setting of 1.728GHz is represented by the grey shaded area under the distribution, and the likelihood of meeting that deadline is 95.48%. The data also presents tradeoffs between likelihood and PPW. While a frequency of 1.728GHz provides a high likelihood of satisfying the deadline, it is suboptimal in terms of PPW (see figure 9(b)). If the a frequency of 960 MHz was selected, the likelihood is much less (55%) but the PPW is close to the optimal (0.315). The reduced likelihood implies the possibility of the image quality being degraded.

Figure 10 shows this result. It's clear that as the frequency

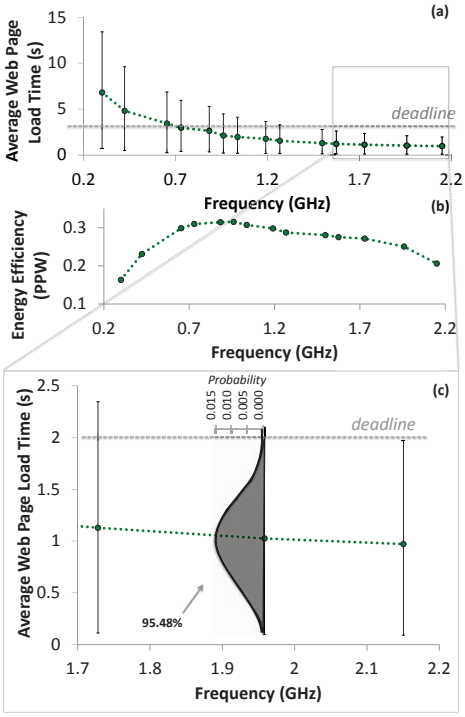


Figure 9: (a) Execution time, (b) energy efficiency results at all available frequencies for *web browsing* on the Dragonboard. (c) Example calculation of the probability of meeting the specified deadline in 9(a).

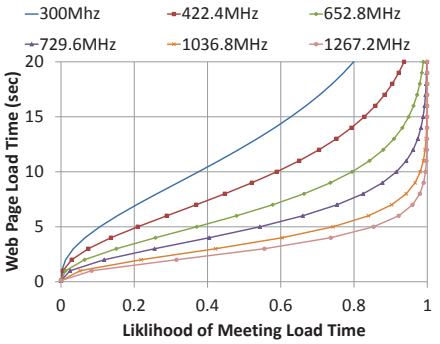


Figure 10: Web page load time versus the likelihood of reaching those load times for various frequency settings.

increases, the variance in web page load times decreases, and the difference in mean execution time reduces. There are diminishing returns on average web page load time; however, the likelihood of reaching any given deadline increases as well. When coupled with the non-linear power model, the data shows that the optimal point may not necessarily be one which gives a necessary level of QoS.

7 Related Works

7.1 DVFS-based Energy Efficiency Control

Over the past decade, a large body of research has been published on optimizing energy efficiency; however, most if not all of the works have assumed that the underlying execution time quantities are deterministic. While there is an extensive amount of prior work on using the available DVFS control knobs to optimize for processor energy efficiency, this work takes into account the probabilistic nature of execution times to ensure performance QoS leading to improved user experiences, that is particularly important for smartphone applications. Furthermore, this work that demonstrates how real-device per-core DVFS control feature can be used to improve the mobile user experience by balancing performance and energy with probabilistic guarantee on QoS.

We summarize some of the most closely related works to this paper [7, 10, 12, 13, 16, 18, 20–24, 26, 27, 36, 38, 43, 44, 47, 52]. Isci et al [27] considered DVFS as a means to dynamically tune a processor's power consumption based on an application's computation and memory characteristics. Others considered thermal constraints and leakage power when optimizing energy efficiency [10, 13, 20, 22–24, 28, 30, 31, 33, 35, 47]. The control-theoretic techniques, e.g. [7, 23, 26, 36], for multicore processors usually involve a model-predictive controller to determine optimal control for one or more control time steps in the future, by solving a constrained optimization problem. A significant contribution to extending the traditional DVFS-based multicore controller is adding deadline constraints. For example, a few recent works [12, 21, 22] proposed learning algorithms for the temperature and power models. However, these works assume deterministic runtimes for applications; therefore, deadline constraints are only a deterministic function of processor core frequency.

While most of the prior works focus on optimizing system energy efficiency for the application domain of CMPs, a few more recent works started examining and designing energy efficiency optimization algorithms for user-centric, interactive smartphone applications. Egilmez et al. [14] specifically aimed at improving user satisfaction through DVFS-based control knobs to maintain a low and comfortable device skin temperature. Singla et al. [50] developed power and thermal models for a modern mobile platform and proposed a closed loop thermal control to adjust processor frequencies.

Zhu et al. [52] proposed an event-based scheduling approach to improve the energy efficiency of web browsers for smartphones while meeting specified QoS constraints. Their method displayed a 33% improvement in energy efficiency over the linux Interactive governor; however, this work only targeted web browser workloads. Additionally [39] found that with detailed knowledge of the application domain an average of 16% energy efficiency improvement can be achieved over the existing linux governors.

Another very relevant work appears in [32]. The authors characterized the tradeoffs between power management and tail latency for server applications. From this analysis they determined that the commonly used service techniques were not sufficient to reach the stringent server QoS requirements and did not provide the most energy efficient method of service either. Instead they suggest a careful, workload specific frequency scaling policy. Our work takes advantage of this insight by creating workload specific performance models which help identify the optimal frequency setting.

The previous works have demonstrated that additional application knowledge can yield higher energy-efficiency results. As such, the probabilistic guarantee on performance QoS proposed in this paper (Section 3) can be used to guide existing dynamic energy and thermal management techniques for applications with pre-specified deadlines. This paper has demonstrated an average energy efficiency improvement of 34% over existing linux governors and displays this over a larger variety of applications than previously mentioned works.

7.2 Performance Modeling and Analysis

It is common to represent tasks bound to processing elements as a collection of linear equations based on the ex-

pected throughput of each task [5, 40, 49]. These methods are advantageous in their simplicity; however, they typically overlook crucial characteristics of the system and application which can lead to inaccurate results. To overcome some of the limitations caused by the use of linear equations, Bogdan et al. [9] proposed a performance prediction model which leveraged fractional calculus. This extension proved to be effective in deterministic routing systems. Another common methodology for performance analysis involves sampling. Similar to the problem setup of this paper, Wernsing and Stitt [51] suggested that the execution time of a task is a function of some work metric or characteristic. In this case the authors limited this function to a deterministic, monotonically non-decreasing function which is determined via regression analysis. From these models, the expected function execution time is directly predicted based on the work metric and proved to be an effective means of performance prediction in systems with low execution time variance.

To model pipelined parallel applications, scenario-aware dataflow models offer an accurate albeit, complex solution [11]. Real-time calculus and network calculus [15] offer an alternative which provides some solutions to the complexities of scenario-aware dataflow modeling. For example, a round robin scheduler which used real-time calculus for data flow prediction was developed for image processing on multiprocessor system-on-chip [46]. Qian et al. proposed a network calculus performance model for a general multi-router system with contention [45]. However, these prior network calculus-based works assumed that the system is a deterministic queuing system.

On the other hand, this work

- considers non-determinism in the architecture and application software,
- formally characterizes and expresses the execution time as a statistical distribution and propose the resulting QoS constrained energy efficient optimization problem, and
- performs an in-depth energy-efficiency evaluation on a real-system platform and demonstrate a use case for quantitative measure of the probability of meeting an execution time deadline.

8 Conclusion

This work provides a new and necessary step toward developing a methodology for optimizing energy efficiency subject to user satisfaction. We develop a statistical framework for characterizing, profiling, and predicating the execution time of parallel applications running on mobile platforms.

The proposed statistical framework provides additional and valuable information that can be used to guide the control of voltage/frequency settings available on modern processors. This is particularly helpful for today's energy-constrained smartphones that execute real-time apps with execution time constraints. With the detailed performance and energy efficiency characterization on web browsing, we show how the proposed statistical framework is used to consider the energy efficiency for smartphones while accounting for a probabilistic guarantee on QoS.

Specifically, the optimization method creates a set of static tables containing frequencies corresponding to optimal energy efficiency, subject to specified QoS levels. Once the application is started, the processor state is altered based on the desired QoS level. This is an important first step towards ensuring user experience in non-deterministic workloads. However, we recognize that there is significant room for expanding the work. For example, the most desirable case would be to dynamically tune the processor based on the changing characteristics of the user, input, background task noise, and other system state. Despite our static nature, our proposed optimization method achieved a 29% power savings over commonly used Linux governors while maintaining an average web page load time of 2 seconds with 90% likelihood—a quality that the Linux governors do not consider for user-centric mobile applications.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful feedback and Qualcomm Research for the generous equipment donation and tool support. Additionally, we would like to thank David Brooks and Margaret Martonosi for their time reviewing this work. This work was partially supported by the NSF I/UCRC Center for Embedded Systems (NSF grants 1361926, 1432348) and by Science Foundation Arizona under the Bisgrove Early Career Scholarship. The opinions, findings and conclusions or recommendations expressed in this manuscript are those of the authors and do not necessarily reflect the views of the Science Foundation Arizona.

References

[1] "Android NDK," https://developer.android.com/tools/sdk/ndk/index.html, accessed: 2014-8-11.
[2] "Mozilla firefox." [Online]. Available: https://www.mozilla.org
[3] "Videolan organization." [Online]. Available: http://www.videolan.org
[4] "How loading time affects your bottom line," https://blog.kissmetrics.com/loading-time, 2011, Accessed: 2015-05-14.
[5] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini, "A feedback-based approach to dvfs in data-flow applications," *the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 11, pp. 1691–1704, 2009.
[6] M.-S. Alouini, A. Abdi, and M. Kaveh, "Sum of gamma variates and performance of wireless communication systems over nakagami-fading channels," *Proceedings of the IEEE Transactions on Vehicular Technology*, vol. 50, no. 6, pp. 1471–1480, 2001.
[7] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, march 2011, pp. 1–6.
[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.
[9] P. Bogdan, R. Marculescu, S. Jain, and R. T. Gavila, "An optimal control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly variable workloads," in *Proceedings of the Sixth IEEE/ACM International Symposium on Networks on Chip*, 2012, pp. 35–42.
[10] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 171–182.
[11] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Throughput-constrained DVFS for scenario-aware dataflow graphs," in *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, 2013, pp. 175–184.
[12] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2007, pp. 207–212.
[13] J. Donald and M. Martonosi, "Techniques for multicore thermal management: Classification and new exploration," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006, pp. 78–88.
[14] B. Egilmez, G. Memik, S. Ogrenci-Memik, and O. Ergin,

"User-specific skin temperature-aware dvfs for smartphones," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 2015, pp. 1217–1220.
[15] M. Fidler, "Survey of deterministic and stochastic service curve models in the network calculus," *Communications Surveys & Tutorials, IEEE*, vol. 12, no. 1, pp. 59–86, 2010.
[16] K. Flautner and T. Mudge, "Vertigo: Automatic performance-setting for linux," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.
[17] C. Gao, A. Gutierrez, M. Rajan, R. Dreslinski, T. Mudge, and C.-J. Wu, "A study of mobile device utilization," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2015.
[18] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas, "Policies for dynamic clock scheduling," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, 2000.
[19] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2011, pp. 81–90.
[20] V. Hanumaiah, S. Vrudhula, and K. S. Chatha, "Performance optimal online DVFS and task migration techniques for thermally constrained multi-core processors," *IEEE Transactions on Computer-Aided Design*, vol. 30, no. 11, pp. 1677–1690, November 2011.
[21] V. Hanumaiah, D. Desai, B. Gaudette, C.-J. Wu, and S. Vrudhula, "STEAM: A Smart Temperature and Energy Aware Multicore Controller," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 5s, Sept. 2014.
[22] V. Hanumaiah and S. Vrudhula, "Temperature-aware dvfs for hard real-time applications on multicore processors," *IEEE Transactions on Computers*, vol. 61, no. 10, pp. 1484–1494, 2012.
[23] ——, "Energy-efficient operation of multicore processors by dvfs, task migration, and active cooling," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 349–360, 2014.
[24] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "A framework for dynamic energy efficiency and temperature management," in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 202–213.
[25] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2014, no. 325462-052US.
[26] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of the International Symposium of Microarchitecture*, 2006, pp. 347–358.
[27] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 359–370.
[28] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st Annual Design Automation Conference*, 2004.
[29] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik, "Parallelizing the web browser," in *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.
[30] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz, "A control-theoretic approach for energy efficient CPU-GPU subsystem in mobile platforms," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, p. 62.
[31] A. Kahng, S. Kang, R. Kumar, and J. Sartori, "Enhancing the efficiency of energy-constrained DVFS designs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 10, pp. 1769–1782, Oct 2013.
[32] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *IEEE International Symposium on Workload Characterization*, 2014, pp. 31–40.
[33] K. Kang, J. Kim, S. Yoo, and C.-M. Kyung, "Temperature-aware integrated DVFS and power gating for executing tasks with runtime distribution," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 9, pp. 1381–1394, Sept 2010.
[34] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer Science & Business Media, 2001, vol. 2050.
[35] J. S. Lee, K. Skadron, and S. W. Chung, "Predictive temperature-aware DVFS," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 127–133, Jan 2010.
[36] W.-Y. Liang, P.-T. Lai, and C. W. Chiou, "An energy conservation DVFS algorithm for the android operating system," *Journal of Convergence*, vol. 1, no. 1, pp. 93–100, December 2010.
[37] W. Liao, L. He, and K. M. Lepak, "Temperature and supply voltage aware performance and power modeling at microarchitecture level," *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1042–1053, 2005.
[38] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2001.
[39] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Domain knowledge based energy management in handhelds," in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, 2015, pp. 150–160.
[40] U. Y. Ogras, P. Bogdan, and R. Marculescu, "An analytical approach for network-on-chip performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 12, pp. 2001–2013, 2010.
[41] A. S. V. Palladi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium*, vol. 2, no. 223-238, 2001, pp. 3–3.
[42] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite-mobilebench," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2013, pp. 133–142.
[43] T. Pering, T. Burd, and R. Brodersen, "Voltage scheduling in the lpARM microprocessor system," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.
[44] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.
[45] Y. Qian, Z. Lu, and W. Dou, "Analysis of communication delay bounds for network on chips," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, 2009, pp. 7–12.
[46] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst, "Reliable performance analysis of a multicore multithreaded system-on-chip," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2008, pp. 161–166.
[47] J. S. Seng, D. M. Tullsen, and G. Z. Cai, "Power-sensitive multithreaded architecture," in *Proceedings of the 2000 International Conference on Computer Design*, 2000, pp. 199–206.
[48] D. Shingari, A. Arunkumar, and C.-J. Wu, "Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2015, pp. 22–33.
[49] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8, 2012, pp. 11–22.
[50] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 2015, pp. 960–965.
[51] J. R. Wernsing and G. Stitt, "Elastic computing: A portable optimization framework for hybrid computers," *Parallel Computing*, vol. 38, no. 8, pp. 438–464, 2012.
[52] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, 2015, pp. 137–149.
[53] Y. Zhu and V. J. Reddi, "Webcore: architectural support for mobileweb browsing," in *Proceeding of the 41st annual international symposium on Computer architecuture*, 2014, pp. 541–552.