NORTHWESTERN UNIVERSITY

User-aware System Design and Optimization

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering and Computer Science

By

Matthew Schuchhardt

EVANSTON, ILLINOIS

August 2015

ABSTRACT

User-aware System Design and Optimization

Matthew Schuchhardt

Humans are incredibly diverse and complex. Unfortunately, personal computing devices are designed and optimized to best suit the average user, rather than considering the individual. Over-designing a computer system is wasteful from a power and expense standpoint. On the other hand, under-designed systems use less power and hardware, but can harm the overall user experience by not successfully meeting the user's performance needs. This leaves a significant amount of room for optimization and improvement in the design of these systems; without a full understanding of the user impact of a design decision, any solution is likely to be suboptimal.

To address this, this work proposes the use of systems which have an improved contextual understanding of the individual user. This not only enables the creation of a more personalized system, but also allows the system to react to fluctuations in individual's requirements of the system over time. Additionally, this work explores the benefits to be gained by targeting performance metrics which are known to closely reflect actual user sentiment when designing and optimizing hardware and systems.

The overarching goal of this work is to provide an argument against the old mantra of "one-size-fits-all" computers, and to help motivate systems which are able to better suit a diverse set of users.

Acknowledgements

Over the course of preparing this dissertation, I have interacted and collaborated with many individuals, without whom, this work surely would not have been possible. Hence, I would be remiss to not show my appreciation for their contributions, both to my research, and also to other facets of my life over the past several years.

First of all, I want to thank my advisor, Prof. Gokhan Memik, for welcoming me into to his lab, and for all of his assistance and guidance over the years. I feel that I have grown immeasurably as a researcher and computer engineer, and I owe a large part of that growth to the time that I spent working under him. By his encouragement, I was able to push beyond my comfort zone of expertise, and acquired a much broader understanding of my field as a result.

I also wish to thank the faculty, researchers, industry members, and students that I have worked with, and for allowing me to share in their knowledge and expertise. Prof. Peter Dinda offered me a unique perspective on computing systems, and was an ever-present source of guidance. In that same vein, I want to thank the rest of the members of the Empathic Systems Project and the many labmates and students that I have interacted with. Further thanks go to Intel's Strategic CAD Labs for hosting me for my two (extended) summer internships — especially to Michael Kishinevsky, Susmit Jha, and Raid Ayoub, for collaborating on many of my published papers, for participating on my defense committee, and for their extensive technical insight.

I also want to thank my wife, Hannah, for all of her love and support during my PhD studies. Deadlines and stressful nights were far more surmountable with her help over the years, and I hate to think how much more difficult my PhD would have been without her in my life. Finally, I want to thank my parents Gregg and Sue, and my siblings Scott and Sarah. Their love and encouragement has been a constant source of inspiration from a very young age, and I truly appreciate all of the countless opportunities they've given me.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The diversity that humans exhibit is remarkable. Physical differences are the most immediately apparent, but when you also consider other variations among people — backgrounds, abilities, personalities, passions, preferences — indeed, no two people are alike. With that in mind, it's interesting that the computing industry still largely designs their systems under the banner of "one-size-fits-all". When a manufacturer designs a new device model (be it a smartphone, tablet, or a more traditional computing device), all of those models are shipped with the same performance characteristics, power curves, display attributes, font sizes, application settings, and so forth. Meanwhile, users have varying sensitivity to performance characteristics and device usage patterns [117, 40, 35, 128, 34]. Of course, some systems are configurable to an extent, but with the industry's heightened focus on the user experience, asking users to manually configure their systems is inconvenient at best, and unused at worst [28, 118, 111].

Ideally, every computing device would be customized to suit each user's needs. Computer engineers might appreciate the number of jobs this would create, but most consumers would balk at the notion of a multi-million dollar, custom-built smartphone. In reality, over-provisioning is commonly used to suit the most demanding consumers, but this additional performance sacrifices power efficiency in users who don't require or care about the extra performance. In lieu of that, in this work I stress the importance of computing systems which are able to better understand and react to user requirements, rather than treating all users as the same. Furthermore, I discuss the importance of designing and optimizing systems with metrics that closely align with user preferences. From this, the end goal is that personal computing can begin to actually feel personal.

1.1. User variability and context

As was already alluded to, no two computer users are the same, and this variability between users is reflected in differing requirements in computing devices.

For example, let's consider the core of any personal computing device — the CPU. Modern CPUs are able to run at a variety of frequency levels [102], which allows the processor to sacrifice some performance to consume less power. In isolation, the CPU would ideally run at 100% frequency at all times; this would provide the maximum amount of performance possible (ignoring thermal throttling requirements). However, this would be quite energy-inefficient; not only do workload performance requirements vary [60, 83], but even individual users have varied responses to throttled frequency levels [34, 84, 116]. An optimal solution would be fully aware of the workload and user's performance requirements, and optimize the system power with that constraint in mind.

This can be extended beyond CPU frequency. Consider visual acuity, which is a metric that describes how well a person can see [67]. Users with poor visual acuity have difficulty resolving visual detail. Hence, in the computing realm, they either require a larger display, or larger details on the display. Additionally, users with poor visual contrast sensitivity require brighter screens (effectively boosting the screen content's contrast) to see the same level of detail [56]. Similar to the CPU example, a display running at maximum brightness is not power efficient (and could actually cause eye strain [94]). A user-aware system would be able to understand what each individual's needs are, and provide an optimized configuration.

It's even possible for a single user to have requirements which vary depending on their own state and the environment that they are in [128]. I refer to this combined internal and external state as *context*. This means that not only should personal computing devices be tailored to the individual, it should also attempt to better serve what the user requires at any given point in time. Predicting context can be extremely complex, as there are many interacting variables, but beginning to understand user context can make great strides in better serving user requirements.

To examine the implications of this variability in requirements, Chapter 3 examines the system performance requirements that users have in an interactive video game. I specifically investigate preferences regarding various levels of CPU frequency, CPU utilization, and screen brightness, and consider the interplay between them. Ideally, it would be possible to simply run the system at maximum performance constantly, but if users don't require this level of performance, this is a inefficient use of power. During this study, my findings support the fact that user requirements vary significantly. Furthermore, I use an array of biometric sensors and an online learning algorithm in an attempt to predict how satisfied the user is with the system. Being able to understand the user's emotional state without any direct invention from the user allows us to non-invasively control and optimize system state.

In Chapter 4, I examine display brightness preferences on smartphones. Because of the wide-ranging environmental lighting situations that users operate their smartphones in, many manufacturers are now including ambient light sensors on the devices. These light sensors automatically dim the display's brightness in low-lit environments, and vice-versa. However, I find that users have widely varying brightness requirements. Furthermore, I find that users' preferred brightness levels not only depend upon the ambient light levels, but also upon other contextual information, such as remaining battery life or time of day (among others). By using an online model, I am able to more accurately control the display's brightness, and also provide an individualized experience without requiring explicit configuration settings.

1.2. User-facing metrics

The computer engineering community as a whole is obsessed with performance. This is by no means a bad thing; the entire industry is built around the notion of continually improving the speed, quality, and/or efficiency of the various computing subsystems. This focus is directly responsible for the tremendous advancement in technology over the past decades.

To guide this constant improvement in computing devices, *metrics* are used to gauge just how well various parts of a computer system operate. Some of these metrics describe the speed of the system — input latency, task duration, throughput. Others explain a device's efficiency — energy per operation, average workload power consumption. Some are more physical descriptions — size, weight, brightness, and even cost. Although there are hundreds of metrics than can be considered in a device as complex as a modern computer, the goal of designing for these metrics is to improve the end product in some way.

Because these metrics are ultimately used to somehow improve a system for a user, let's briefly consider how these metrics might impact a user. Users don't pick up a tablet and say, “wow, the L1 cache latency on this CPU is really bad”. I’ve never seen someone use a smartphone and say, “I like this phone, but this display is 24 nits too dim”. A laptop has never been praised by a consumer as having “really impressive thermal insulation”.

Devices have, however, been described as too slow, hard to read, too warm, difficult to hold, or having bad reception. Before I get completely written off by any engineers reading this, *I am not arguing against the use of performance metrics.* Metrics are an incredibly useful (necessary, even) method of assessing the impact of any system-level change, and will always be part of the design cycle. I am arguing that the implications of applying these metrics must be well-understood to be able to fully benefit the user.

To illustrate this, consider the display brightness example alluded to earlier. Display manufacturers are always trying to get an edge over their competitors, striving to make better device screens. One important performance metric in these displays is the display’s brightness; brighter screens are easier to see, especially in situations with a lot of environmental light, like direct sunlight. However, display brightness comes at a cost; generally speaking, more light requires more electrical power. That is great for users who require the additional screen brightness; however, recall the variety in preferences that users exhibit. An overly bright screen does nothing but consume unnecessary power. To properly balance power consumption with display brightness, the impact that display brightness actually has on the end user must be fully understood; metrics applied blindly are inefficient.

In Chapter 5, the trade-off between display brightness on smartphones and user satisfaction is explored further. Users are asked to rate their satisfaction with display brightness levels in a variety of ambient lighting environments. Furthermore, users complete a task which gauges their ability to see visual detail in these same lighting environments.

This gives a number of interesting results. First of all, I are able to find a strong correlation between an objective visual metric and subjective user ratings. This is useful, because it allows display brightness to be optimized without necessarily having to manually gather opinion ratings from the user. Additionally, I find that there are diminishing returns in display brightness, which allows us to design a system which can reduce display brightness with a minimized average impact on user satisfaction. Better understanding of performance metrics' impact on the end user, in this case display brightness, ultimately allows for designing systems which better serve user requirements.

In addition to designing devices and systems, debugging these systems can be of equally high importance. Fixing traditional functional bugs is "easy" (with the possible exception of concurrency-related bugs). Functional bugs typically only result from the interaction of a small number of devices, and have some degree of predictability. Performance bugs, however, can be much more difficult to detect and fix [96]. Not only are these bugs often intermittent (meaning that they only trigger sporadically), but these bugs can occur due to complex interactions between many hardware and software components. These complex interactions can be difficult to understand the nature of. To make matters worse, these performance bugs directly impact the user experience, which is important in modern interactive devices.

There is some tooling available which is designed to help locate the sources of performance bugs. Profilers, static code analyzers, and dynamic performance evaluators can all help engineers understand the nature of some types of performance bugs. Unfortunately, the types of performance bugs that these systems can help identify are often relatively simple, only covering a small number of abstraction levels. Furthermore, these tools tend to analyze bugs in terms of metrics which don't necessarily have any direct impact on the actual user experience. A performance bug localization system which operates at the system level in a user-relevant manner would alleviate many of these shortcomings.

In Chapter 6, a method for system-level performance bug localization is introduced. This system is designed to run on Android smartphones, and works by analyzing system data and extracting features relevant to performance. This is generalizable to be able to analyze many classes of performance bugs, but this work specifically focuses on the graphics subsystem, due to its direct impact on users and high degree of interactivity. By doing this, if a frame takes a long time to render, the system's state is compared during that frame's rendering to state during properly-rendered frames, to determine the sources of the performance degradation. Furthermore, feature selection algorithms can then be used to automatically compare system state during fast frames to slow ones; by doing this, the most significant contributors to frame rate slowdown can be automatically identified. This method has the advantages of not only debugging the entire system as a whole, rather than just the application, but also directly analyzes frame rate, which is a direct user-facing metric that has a strong correlation with user satisfaction.

1.3. Attribution

The remainder of this work is based on results which appear in the following previous publications:

- M. Schuchhardt, B. Scholbrock, U. Pamuksuz, G. Memik, P. Dinda, and R. P. Dick. *Understanding the Impact of Laptop Power Saving Options on User Satisfaction*

Using Physiological Sensors. In Proceedings of the 2012 International Symposium on Low-power Electronics and Design (ISLPED), pages 291296. ACM, 2012.

- M. Schuchhardt, S. Jha, R. Ayoub, M. Kishinevsky, and G. Memik. ***CAPED:** Context-Aware Personalized Display Brightness for Mobile Devices.* In Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), page 19. ACM, 2014.
- M. Schuchhardt, S. Jha, R. Ayoub, M. Kishinevsky, and G. Memik. *Optimizing Mobile Display Brightness by Leveraging Human Visual Perception.* In Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). ACM, 2015.
- M. Schuchhardt, S. Jha, M. Kishinevsky, and G. Memik. *System-Level Performance Bug Localization Via Causality Analysis.* Under submission.

These publications have all been written with collaborators who were instrumental in the works' completion. Hence, the following chapters use the collective "we" rather than "I", such as to not misconstrue a collaborative effort.

CHAPTER 2

Background

In this section, some ideas and concepts related to this work are presented. These sections are not intended to reflect new research or knowledge by the author, but they are important to be familiarized with before a proper understanding of the remainder of this work can be reached. Section 2.1 describes some important concepts behind the human visual system, and how this applies to electronic displays. Because a significant part of this work was done on the Android mobile platform, we specifically describe some of the more important parts of Android's graphics subsystem in Section 2.2.

2.1. Display Optical Characteristics

Luminance (L) and *contrast ratio* (C) are two of the most important metrics involved with gauging how easily the human visual system can see details on a mobile display [70, 107]. Luminance is defined as the intensity of light per unit area, and is colloquially referred to as *brightness* in the context of mobile displays. Contrast ratio is related to luminance; generically defined, contrast ratio compares the luminance of the colors *black* and *white* (L_{black} and L_{white}) as they appear on the display. L_{black} is never exactly zero on an LCD display; there is always some amount of light which leaks through the liquid crystal matrix, even on an image which is intended to be fully black. Similarly, L_{white} is limited by how bright the LCD's backlight is. The contrast ratio describes the relative difference between these luminances. However, there is not a single well-defined metric which describes contrast ratio, and there are a number of different equations which can represent this. One common formula for a display's contrast ratio is described in Equation (2.1):

$$C = \frac{L_{white} - L_{black}}{L_{white}} \tag{2.1}$$

There are some complexities as to how this is measured and reported in consumer devices (such as dynamic contrast ratio, static contrast ratio, etc.), but as far as light perception is considered, this definition is sufficient. It is important to note, L_{black} and L_{white} are not entirely self-contained to the display itself; the device's ambient light can have a significant impact on these values. For instance, in a completely darkened room, the contrast ratio is precisely defined as in Equation (2.1). However, consider a lighting environment which involves a significant amount of ambient light. Since the display's optical characteristics necessarily don't absorb 100% of incoming ambient light, some of that light is reflected back at the viewer. This reflected light reduces the resulting contrast ratio, as evidenced in Equation (2.2), with E representing the environmental illuminance and ρ representing the display's reflective coefficient. Contrast ratio has a significant impact on discerning visual detail, and this relationship of contrast ratio with ambient light explains why it is easier to read backlit displays in darker ambient environments. This is also the basis for the inclusion of *adaptive brightness systems* in mobile devices. Adaptive brightness systems use ambient light sensors to help control the display's brightness: in dim environments, the display reduces the display's brightness, and in brighter environments, the display is set to higher brightness levels. The net result of these systems is that the display uses less power and avoids eye strain in dim environments, while maintaining better contrast ratio in bright ones.

$$C = \frac{L_{white} - L_{black}}{L_{white} + \rho E} \tag{2.2}$$

Now that contrast ratio and its relationship with environmental light has been introduced, how does contrast ratio and luminance actually impact people's ability to view a mobile display? *Readability* is a metric which describes how easily a person can discern visual detail in a given environment. However, similar to contrast ratio, readability doesn't have a single definition, and can be represented in a number of different ways. We will expand further upon one particular readability metric, Relative Visual Performance (RVP), in Section 5.3.1, but for now, will refer to readability in the generic sense.

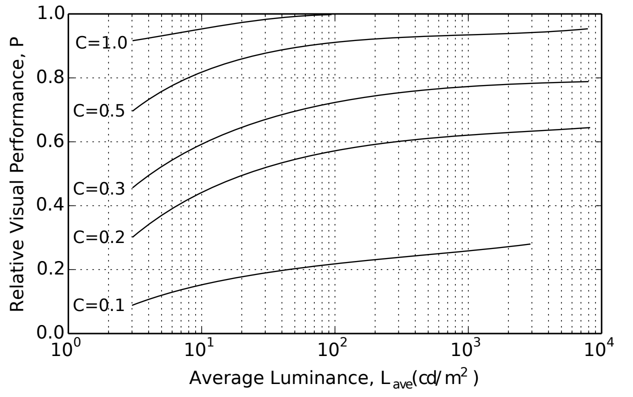
Contrast ratio has a heavy influence on readability. This relationship is intuitive; contrast ratio is similar to the concept of signal-to-noise ratio, where the magnitude of the signal is compared to the magnitude of the noise on top of the signal. Larger differences between these magnitudes leads to a stronger, more obvious signal. Luminance's relationship with readability, however, is somewhat less direct. One might expect that if the contrast ratio is the same at two different luminance levels, the readability would also be the same. However, this is not the case. The human eye is not a simple sensor; there is a homogeneous mix of types of light-detecting cells, each of which respond to different ranges of luminance levels. Because of this, only some of the light-detecting cells in the eye are active in low-light environments, which reduces the eye's effective resolution [67]. In high-luminance environments, all of the light detecting cells remain active, which improves the eye's visual acuity. The end result of this is that readability is better at higher luminance levels, when the contrast ratio is held constant. This doesn't mean that increased ambient light leads higher to readability, however; increasing ambient light increases the screen's luminance due to reflected light, but it also reduces the contrast ratio, which lowers the effective readability on most LCD displays (as Equation (2.2) describes). Contrast ratio is a more significant indicator of readability than the overall luminance [67].

The standards organization CIE [25] conducted a study intended to gauge the impact that contrast ratio and luminance have on readability. Readability was measured across a selection of users, at a number of contrast ratios and luminance levels, the results from which are shown in Figure 2.1. As previously discussed, the study found both contrast ratio and luminance to strongly impact readability. There is also a significant degradation of readability as users age. The curves contain aggregated data from a wide selection of users, and individual users will each have their own personal readability curves, but on average, a 75-year-old requires roughly twice the contrast ratio that a 20-year-old does at the same luminance and contrast ratio to achieve similar readability levels. This trend in readability data is further explored by Kelley et al. [70].

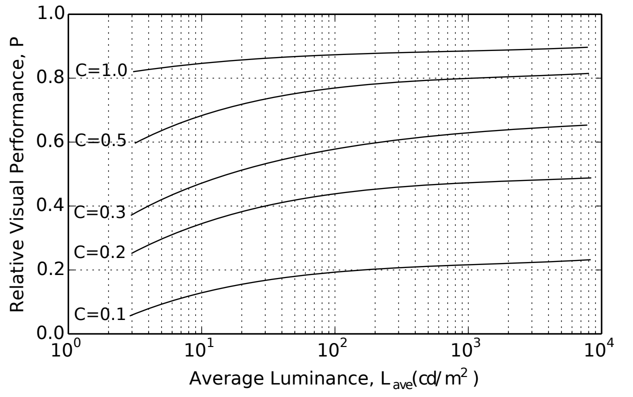
2.2. Android Display Subsystem

We first describe how the modern graphics architectures are designed, and then describe the specifics of Android's graphics system.

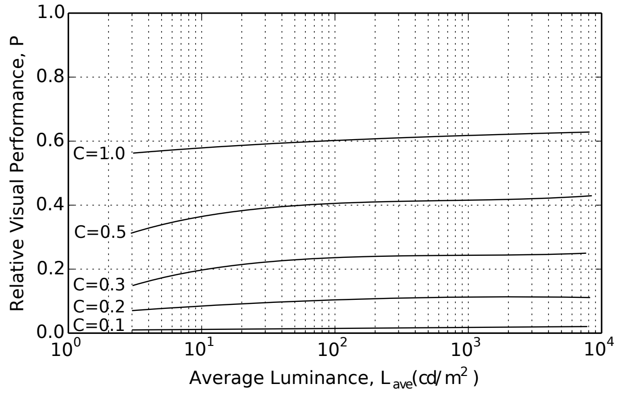
In most modern devices, the graphics subsystem is based around the notion of a framebuffer. A framebuffer is an area of RAM (either in system memory or as part of the



(a) 20 year old, 1.5' detail



(b) 50 year old, 1.5' detail



(c) 75 year old, 1.5' detail

Figure 2.1. Aggregated readability curves [70] for users of varying ages, generated using data from CIE [25]. As users age, their ability to discern detail drops significantly at a given luminance and contrast ratio level. Contrast ratio and luminance are both strong predictors of readability.

video device) which holds the bitmap for the image that the output display shows. Any hardware with access to this area of RAM can modify this memory region, be it the CPU, GPU, or a coprocessor; when the memory is modified, the image shown on the display changes accordingly. However, displays don't update their displayed image immediately when the framebuffer is changed. In reality, the display hardware pulls the data from the framebuffer memory and then shows the newly updated image at a constant refresh rate. 60Hz is currently a common refresh rate for LCDs, but higher rates are becoming increasingly prevalent.

Because the video RAM can be arbitrarily written to, and the display updates the image at a regular interval, it is actually possible for the display to show a frame which is only partially rendered. This phenomenon is known as *screen tearing.* Screen tearing typically appears as an image with a discontinuous horizontal line, where the top half is part of the newly rendered frame, and the bottom half is part of the old frame. Even though this discontinuous image only appears for the duration of a single frame, it is readily noticed by users as a graphical artifact [5]. To avoid screen tearing, many displays indicate to the operating system when it is about to begin showing a new frame. This signal, known as a V-sync signal (vertical sync, a term carried over from the analog display era), allows the operating system to avoid tearing frames by synchronizing their rendering with the display. Some upcoming displays [5, 46] support a variable refresh rate, which allows the video driver to signal the display when to update its image, avoiding the need for a V-sync signal, but these are not yet implemented in Android and are orthogonal to this work.

This single-buffered system works well in simple situations, but unfortunately, in more complex workloads it's often the case that the system can't render a given frame within the display's refresh time period. With a single framebuffer, it is sometimes impossible to avoid screen tearing, because the display would switch to a partially rendered frame at some arbitrary point in time. To avoid this, many systems employ a graphics system which contains memory space for two framebuffers. At any given point in time, one of the buffers contains a *front* buffer, and the other contains a *back* buffer. The front buffer contains the image currently being displayed on the screen, and the back buffer is used to hold the image currently being rendered. When a V-sync signal comes in, the operating system checks if the back buffer is finished being rendered. If it is ready, the buffers are *flipped*, and the front buffer becomes the back buffer and vice-versa. If the back frame is not finished being rendered, the pages are not flipped, and the front buffer is continued to be displayed until the next V-sync signal; this results in a *dropped frame*, because the front buffer was displayed for two V-sync periods in a row.

As a performance optimization on top of double buffering, some systems also use triple buffering. Triple buffering not only employs a front and back buffer, but also uses a third buffer. After the back buffer is done being rendered (but before the frame flip occurs), this third buffer allows the system to immediately begin rendering on the third buffer instead of having to wait for the page flip to occur. This results in frame rate improvements by allowing more rendering time in certain situations. Android is always triple-buffered by default. Figure 2.2 contains a representation of a buffering system over time. This execution depicts situations with double and then triple buffering being employed. The

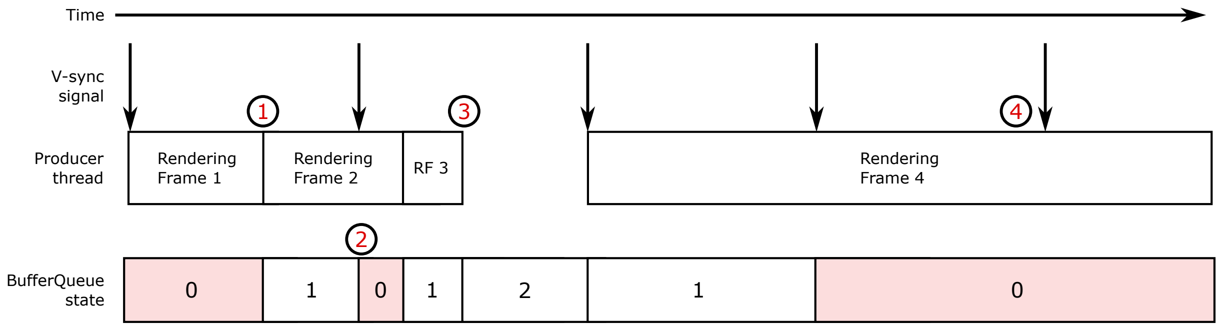


Figure 2.2. Timing of frame rendering and buffering in Android's display subsystem. At ①, a producer has completed rendering Frame 1 and added it to the corresponding BufferQueue. This frame remains in the BufferQueue until the next V-sync signal arrives at ②. At this point, the buffer is dequeued and the frame is shown on the display. Multiple simpler frames are rendered and added to the BufferQueue until full at ③, at which point rendering must pause because no video memory is available to work in. Frames are continued to be dequeued until a V-sync signal arrives with an empty BufferQueue at ④. This results in a dropped frame and appears as a reduced frame rate.

rendering for Frame 4 takes a long time to complete; eventually, a V-sync signal occurs during an empty buffer and causes a dropped frame.

Now that the design of a typical graphics subsystem has been described, we describe exactly how these apply to Android's graphics architecture. The primary blocks in Android's display subsystem are outlined in Figure 2.3. First of all, the BufferQueue is a low-level data structure of particular importance to the graphics subsystem. BufferQueues handle the buffering of graphics data, taking input from a producer source, and buffering that data to eventually be used by a consumer.

The consumer of these buffers is Surfaceflinger, which is Android's window compositor. Surfaceflinger performs many critically important functions in the display subsystem. First of all, Surfaceflinger tracks the location and geometries of the BufferQueues that are currently active on the display. In most situations, the image seen on the display

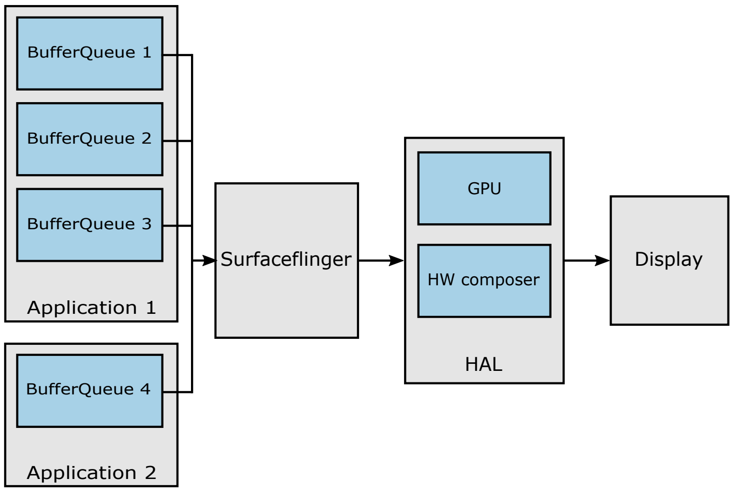


Figure 2.3. Android's display subsystem. Individual segments of frames are produced by applications and added to BufferQueue structures. Surfaceflinger handles V-sync signal synchronization and controls how frame composition is conducted.

is actually multiple BufferQueues which make up different segments of the image. For example, common buffers include the navigation bar, the status bar, video sources, and the Android GUI elements. Because multiple BufferQueues are often active on the display at any given point in time, Surfaceflinger controls how they are composited together into a single resulting image — either via the HW compositor (more efficient) or the GPU (if there are a high number of active BufferQueues). Surfaceflinger also handles the z-ordering and transparency of the BufferQueues, which determines which buffer lies at the top, in the case of multiple overlapping buffers. Secondly, Surfaceflinger handles the timing and synchronization of frames on the display. Whenever Surfaceflinger receives a V-sync signal from the display, it checks all of the BufferQueues to see if they have any new graphics ready to be composited. If no new data is ready in the buffers, then the previous frame is displayed again. If any (or multiple) buffers have new data to display, then the composition is performed and the new frame is displayed.

CHAPTER 3

Understanding the Impact of Laptop Power Saving Options on User Satisfaction Using Physiological Sensors

3.1. Introduction

The primary goal of a computer system is to satisfy the end-user. Traditionally, computer designers measure the effect of their optimizations using metrics expected to correlate with user satisfaction (e.g., tasks per time, instructions per cycle, packets per time). However, previous work has shown that (1) providing higher performance levels does not necessarily improve the user satisfaction and (2) the level of performance necessary to satisfy a particular user varies significantly among users (e.g., [50, 115].) With no knowledge of user satisfaction regarding machine performance, optimizations exercising trade-offs between performance and power consumption are intended to satisfy the nonexistent “typical” user with a “one-size-fits-all” approach. This often leads to sub-optimal configuration choices, which could be improved by understanding individual user satisfaction.

Computer optimizations are typically performed from the perspective of a single component or system block. For example, Dynamic Voltage and Frequency Scaling (DVFS) [91], which reduces power at the cost of CPU performance for a period of time, is controlled independent of screen brightness, which may be adjusted to conserve power. User satisfaction is a function of both CPU performance and screen brightness, but current control schemes do not consider combined effects. This chapter aims to lay the groundwork for a new method of measuring computer performance that would allow the integration of such effects and their derivation from easy-to-acquire user measurements. This new method of evaluating computer systems describes how well a computer performs, not only in traditional terms of raw computational capabilities or power efficiency, but from the perspective of end-user satisfaction.

To accomplish this, we first construct a linear regression-based power model to estimate power consumption of a computer system using software-visible metrics. We leverage this model to estimate the impact of various power-saving techniques on total system power.

To gauge the impact of these power-saving techniques, we perform a series of user studies which involve activating combinations of these techniques during machine use. As we activate these power-saving techniques, we request that users in the study verbally indicate when they are dissatisfied with the computer system. Simultaneously, we use several physiological and behavioral sensors to monitor changes in the user's state. These sensors measure the galvanic skin response (GSR), pupil dilation, pupil movement, wrist movement, the pressure applied to the keyboard during key presses, and the frequency of key presses. The ultimate goal is to find a correlation between machine performance and user biological state.

To draw correlations between computer state and the user's biometric signals, we run a number of analyses. First, we find that the frequency of user-reported dissatisfaction varies by power-saving technique, even when a similar reduction in power consumption is achieved. We also find that poor system performance causes measurable physiological responses; conversely, we find that the sensor readings remain relatively stable when the user is satisfied with the computer's performance. These findings motivate satisfaction-oriented power and performance optimizations in light of constrained power requirements on personal computers.

Because we want to use predicted user state as an input to an eventual device power control system, we need to verify our ability to use this data to make accurate predictions about user state. We devise a prediction system which uses machine learning techniques to classify user state. We divide our data into time slices surrounding each performance reduction event, and label each time slice by a binary user indication. Our system can predict these binary labels with over 80% accuracy, which is a significant improvement over the baseline level; the system also is able to give both good precision and recall. Finally, we perform a supplementary study which demonstrates that our prediction system performs equally well whether users verbally indicate dissatisfaction or not.

The rest of the chapter is organized as follows. In Section 3.2, we present our experimental setup. In Section 3.3, we present our analysis of the sensor data, as well as the results of our prediction system. We provide some final thoughts in Section 3.4.

3.2. Experimental Setup

We now describe the environment in which we carried out our studies, and the studies themselves.

3.2.1. Power Model

Before correlating user satisfaction with power savings techniques, we first develop a model for accurately estimating the power saved by each technique. Techniques may then be parameterized to save roughly equivalent levels of system power.

Our target platform is an IBM Thinkpad T61 with a 2.2 GHz Intel Core 2 Duo T7500 processor and 2 GB DDR2 SDRAM running Microsoft Windows XP. This computer is also the platform used during user studies.

The predictors we target in the power model follow: RAM bus transactions, disk accesses, processor utilization percent, screen brightness, screen on/off state, and WiFi on/off state. We also directly measure the power being used by the system; this is our dependent variable.

One of the most common techniques for saving power is reducing CPU frequency. However, changing the CPU frequency substantially alters the amount of power that the other predictor variables would contribute to the power model. Hence, we generate separate power models for the system running at 1.6 GHz and 2.2 GHz.

To create the power model, we run a variety of benchmarks, each which primarily stresses one component on the system. We then create a linear regression model from this data.

To verify this model, we also perform four user-driven verification workloads. Using these workloads, we compare the actual power readings with the power levels predicted by our model. Averaged across all readings and verification workloads, our mean absolute error is less than 2%.

3.2.2. Physiological Sensors

In this work, we utilize a GSR sensor, a 3-axis accelerometer, an eyetracker, and piezoresistive force sensors to measure the physiological and behavioral effects of different power saving techniques. Additionally, we also record keyboard button presses via software hooks.

Although they measure a complex biological system, GSR sensors are quite simple in application. GSR sensors measure the skin's resistance to an electric current. Most commonly, GSR sensors consist of two contacts placed on two of the users' fingers on the same hand (although there are alternative locations where the signal can be read, such as the wrist or even the foot Healey [57]). A voltage divider is then used to allow the skin's resistance to be determined with a voltage probe. This variation in resistance is correlated with the with the sympathetic nervous system, and changes with external stress and stimulation [32].

Eyetracking technology can also be used to gather information about the user's emotional state [12]. Pupil dilation and eye movement are both correlated with stimulation and stress responses, and so can give an indication of the user's emotional state. We also collect data on user motion [18, 31], keyboard button press force [81], and keyboard press frequency [81] in an attempt to draw similar correlations.

In our specific setup, the GSR's ADC and the accelerometer are both mounted on a wrist strap, the signals from which are fed into an Arduino-based microcontroller [8]. The GSR contact pads themselves are placed on the small and index fingers of the user's right hand. The force sensors are attached to the arrow keys on our keyboard, and their signals are also fed into an Arduino device. The eyetracker is made by MobileEye [93], and consists of a head-mounted camera which is controlled by a dedicated laptop.

From those sensors and logging software, we extract seven different metrics:

- AccelMag: sum of squares of X, Y, and Z accelerometer axes
- DeltaGSR: change in GSR value since the last sensor reading
- Keypress: time since the last keyboard button press
- MaxForce: largest current value from the force sensors
- NormalMaxForce: same as MaxForce, but normalized to each key's highest force reading
- PupilMovement: change in position of the pupil since the last pupil reading
- PupilRadius: the radius of the pupil.

These metrics are chosen based on their expected correlation with user frustration. For instance, GSR sensors tend to exhibit a low-frequency change in sensor values on top of the more interesting high-frequency signal, so we only look at the change in GSR since the last sensor reading to reduce the low-frequency effects. This is supported by the work from Figner and Murphy [42] and Setz et al. [112], which shows a typical GSR stimulation instance consisting of a sudden change in skin conductance followed by a slow return to its base conductance level.

3.2.3. User Studies

In this section, we describe the techniques that are used to reduce power consumption. To simplify the discussion, we call these techniques *annoyance events*, because they are applied for short durations during our user studies, and they may cause user dissatisfaction/annoyance.

From the results of the power model, we decided to focus on the CPU utilization (CPUUtil), CPU frequency (CPUFreq), and screen brightness (ScreenDim and ScreenGrad). CPUUtil limits the scheduling time of the target application without changing the frequency. CPUFreq reduces the frequency of the system's processor. ScreenDim reduces the brightness of the screen to save power. ScreenGrad does the same, but does so gradually over 10 seconds. The rest of the predictors either would completely disrupt the experiment (e.g., WiFi on/off), or are not easily throttleable (e.g., RAM accesses).

We select a variety of different annoyance events which total up to 4, 8, or 12 Watts of power savings for our target system (Table 3.1). If multiple annoyances are in effect at the same time, the power savings is split between them (e.g., if 3 annoyances are active for 4 Watts of total savings, each annoyance saves 1.33 Watts). We also include *pseudo annoyances*, where no annoyance is initiated by our system. We initiate each annoyance event twice during user studies. Every annoyance event is active for 30 seconds after which the computer reverts back to the original state for 25–40 seconds. The order of annoyance events are randomized to eliminate any possibility of biasing.

The workload that each user in this study performs is the racing game *Motocross Madness 2*. We use a racing game as our workload because it is relatively CPU-bound, and requires consistent attention to play. Future work will likely explore additional workloads. The laptop typically consumes 41 Watts of power while running the game without any active annoyance events.

Table 3.1. Annoyance events used in this study. If multiple events are active, each contributes an equal amount to the total amount of power saved.

Active Annoyance Events	Power Saved (W)
None	0
CPUFreq	4
CPUUtil	4
ScreenDim, CPUFreq, CPUUtil	4
ScreenDim	4
ScreenGrad	4
CPUFreq	8
CPUFreq, CPUUtil	8
ScreenDim, CPUFreq, CPUUtil	8
ScreenDim, CPUUtil	8
ScreenDim, CPUFreq	8
CPUUtil	8
ScreenDim, CPUFreq, CPUUtil	12

As the users are playing the game, we request that they verbally notify us when they become dissatisfied. Other methods of reporting dissatisfaction are possible, such as using a physical button or periodically rating machine performance on a scale of values. Simple verbal reports require minimal user action, which is desirable since gameplay is continuous through annoyance events.

We ran this study on 20 users. The studies are conducted in a well-lit room with overhead fluorescent lighting and drawn shades, to minimize the effect that the time of day and screen brightness have on the user's pupil dilation.

The advertisements for the study were placed around various parts of Northwestern's campus, and the respondents were all university students. Of the 20 users, there were 12 males and 8 females. 10 were undergraduate students, and 10 were graduate students.

3.3. Results

We now examine the results of our study. The end goal is to derive a model for predicting user satisfaction from the physiological sensors.

3.3.1. Annoyance Event Analysis

We now analyze and interpret the rates at which users reacted to the various annoyance events. The results from this analysis are presented in Figure 3.1. One interesting piece of information to note from this figure is that different users have very different reactions to the CPUFreq and CPUUtil annoyances. Users very rarely express annoyance when only the CPU frequency is reduced, even though the corresponding CPU utilization reduction saves the same amount of power. Reducing the frequency not only reduces the number of cycles the processor can complete in a given amount of time, but also allows the processor to reduce its voltage, which may explain this discrepancy in indicated annoyance events.

It's also interesting how users readily notice screen brightness changes. Users frequently report annoyance with the system when the screen is dimmed. One notable exception is the 4-Watt ScreenDim + CPUFreq + CPUUtil annoyance: even though there is a ScreenDim

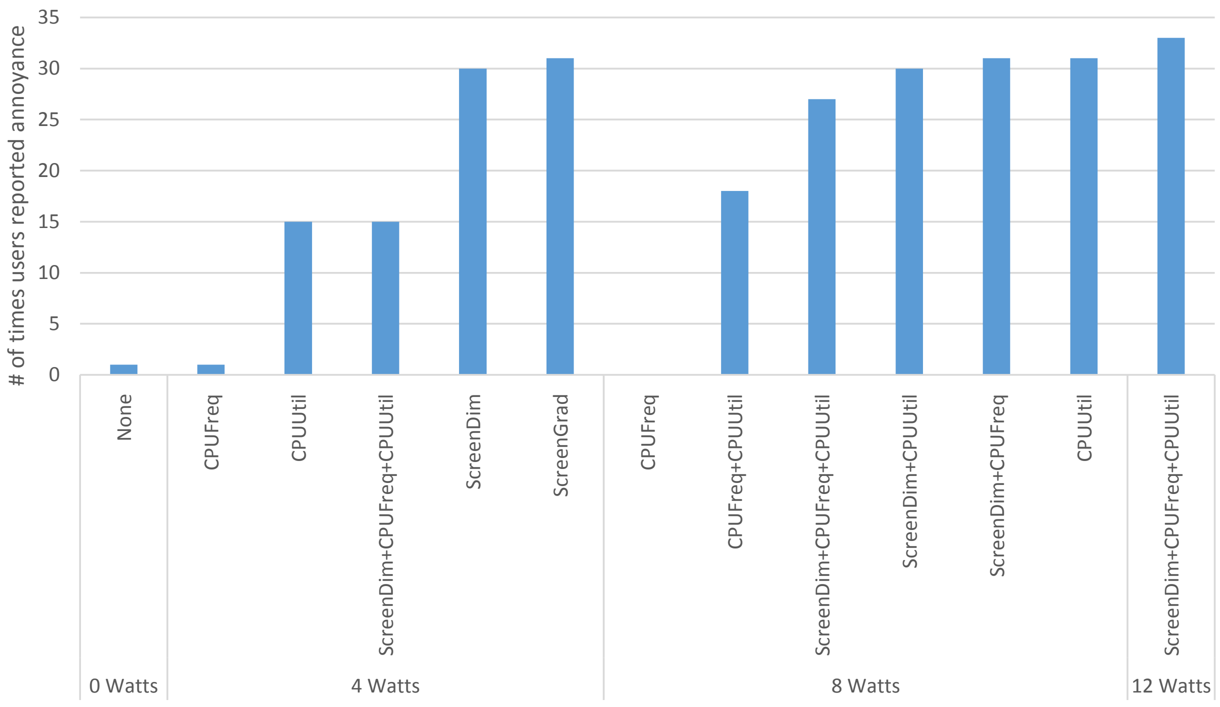


Figure 3.1. Frequency at which users indicated annoyance with a given annoyance event (out of a maximum of 40).

annoyance active in that set, the screen isn't dimmed for this annoyance as much as other cases. This seems to suggest that there is a particular threshold for when users indicate annoyance with a system component. This result alone shows the importance of understanding user satisfaction with power saving techniques: a naïve optimization would dim the screen (with or without a change in CPU frequency) to save power. However, our results show that for this target application, the best method is to partially reduce the CPU frequency, and partially dim the screen. A system that can understand user satisfaction with different optimizations will choose this option over the rest.

3.3.2. Statistical Sensor Value Analysis

To analyze the data collected from the biometric sensors, we first note that there are two distinct sets of data that relates to every individual annoyance event: data during the non-annoyance phase preceding the introduction of the annoyance event, and data during the annoyance phase.

Intuitively, we would expect the sensors to either substantially change before and during an annoyance event, or to remain approximately the same. We hypothesize that events which the users either don't notice or don't care about will not substantially change the sensor metrics. Conversely, we expect that annoyance events which the user perceives as annoying would cause a measurable change between the two sets of sensor readings.

Since the data before and during each annoyance event consists of hundreds of sensor readings, we compute the standard deviation, mean, and median of each of the metrics before and during each annoyance event.

Because each user has a unique reaction to stimuli, we can't know exactly how each users' biometric data will change for each annoyance event. There's a good chance that we would miss some sensor responses if we only looked at one static span of data for each event; some sensors may have a delayed reaction, or the change may happen over a larger or smaller window of time. To improve our likelihood of capturing the timespan that is most indicative of annoyance for each sensor, we consider multiple data windows and offsets to see how the different sensors respond. We look at window sizes of 10, 15, 20, and 25 seconds for multiple offsets from the start of the annoyance event. As a clarification, the larger windows do not have as much room to move within the 25-second window of

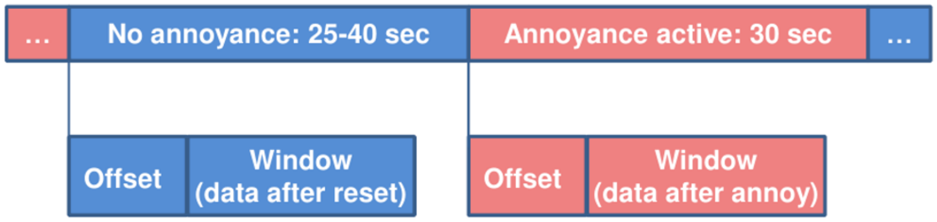


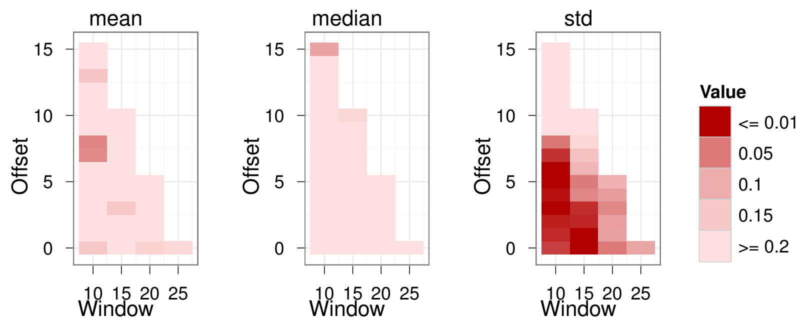
Figure 3.2. Windows & offset scheme used to analyze sensor data surrounding power-saving events. For each power-saving event, data before (blue) and during (red) is compared to identify statistical relationship in sensor data.

non-annoyance, and hence, we do not analyze as many offsets for those windows. This window and offset scheme is illustrated in Figure 3.2.

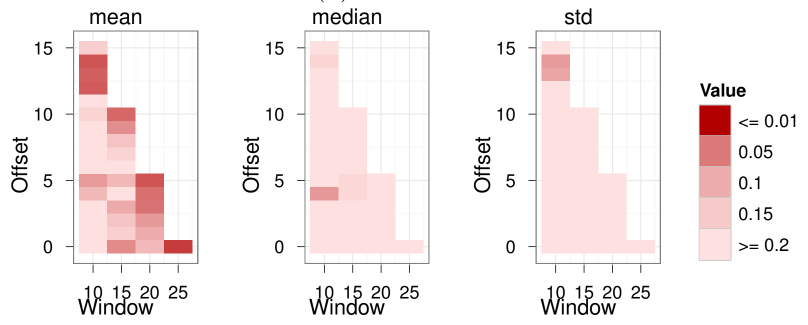
Since we are only concerned with detecting changes in metrics when the user is actually annoyed, we first only consider event instances which are indicated by the user as unsatisfactory. For each event, we have data from seven sensor metrics (Section 3.2.2). For each metric, we collect data from the various window/offset combinations, as described above. On each of those, we then calculate a standard deviation, mean, and median. Each of these data frames consists of the mean, median and standard deviation from the window before and during one annoyance event instance.

All of those similar data frames are then compared from before and during the annoyance using a paired 2-tailed t-test. 2-tailed t-tests with a relevant p-value ($p < 0.1$ in this study) suggest a statistical difference between two sets of samples. Selected sensor metrics are shown in Figure 3.3.

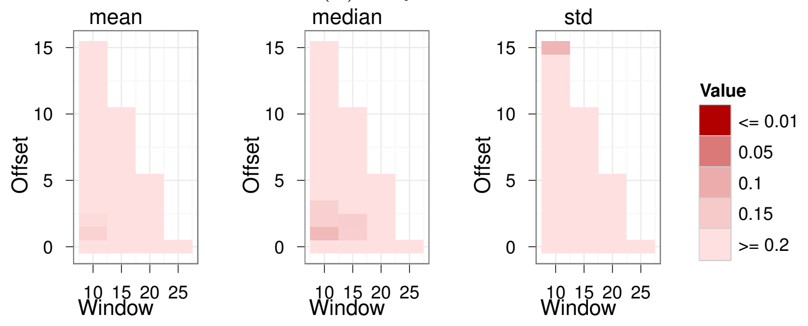
One interesting thing to note in these graphs is that the metrics and windows which give statistically significant t-tests differences vary for each sensor. Some of the sensors respond best with a larger window, while others seem to react best using a smaller one. Additionally, it seems that lower offsets perform better than larger offsets, which would



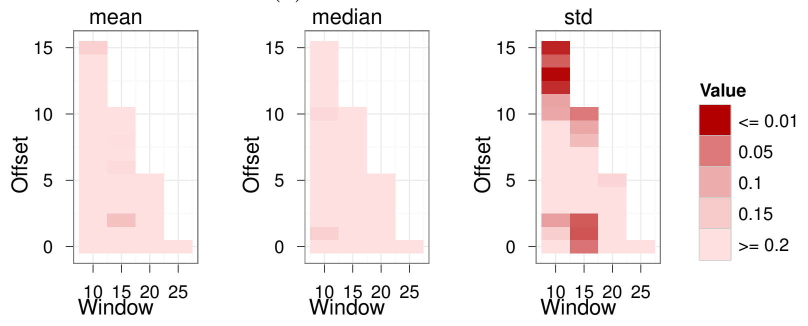
(a) DeltaGSR



(b) KeyPress



(c) NormalMaxForce



(d) PupilMove

Figure 3.3. Windowed t-test analysis for selected sensors. Low p-values, represented by dark areas, indicate metric/offset/window combinations where sensor data differs before and during annoyance events. Only user-indicated annoyances are analyzed.

suggest that these sensors reflect external stimuli fairly quickly. Many of the sensors show promise for our future studies, as we can statistically show the difference between the data before and during an annoyance event with a high degree of confidence for many of the windows and offsets.

3.3.3. Equivalence Analysis

A false positive occurs whenever the user annoyance prediction system indicates that the user is annoyed when they actually are not. A theoretical system could be devised which always detected a difference in some sensor metric. This system would always indicate that the user is annoyed, and so would be 100% effective in detecting annoyance, but would never detect non-annoyance. Thus, we have to prove that our sensor metrics are not only good at detecting differences in sensor data due to annoyance events, but we also must show that they perform reasonably well at detecting non-annoyance. In other words, the system needs both a low false negative rate and a low false positive rate.

TOST [109], or *two one-sided tests*, is one way of demonstrating equivalence between two sets of data. TOST works by running two paired one-tailed t-tests with some offset ϵ.

When proving dissimilarity (Section 3.3.2), we only analyze data sets that the user deems annoying. Conversely, for this equivalence analysis, we only consider data sets in which the user did not indicate annoyance. From the previous t-test analysis, we take the sets of data with the 10 lowest p-values, but at least one and no more than two from each sensor. We then determine the smallest possible ϵ value that is required to achieve a confidence of 90% that the two sets of data are equivalent. In Table 3.2, we present the

Table 3.2. Confidence intervals for equivalence test analysis, using data from the non-annoying events. Only includes selected metrics which performed well in the previous t-test analysis.

Sensor	Metric	Window	Offset	Required ϵ for 90% Confidence	ϵ / avg of means	t-test P-value
AccelMag	std	15	3	0.7048	0.2238	0.1853
AccelMag	std	20	0	0.9376	0.2890	0.0949
DeltaGSR	std	10	5	6.8749	0.1755	0.7906
DeltaGSR	std	10	6	6.9296	0.1764	0.7889
Keypress	mean	25	0	0.0249	0.1297	0.2565
MaxForce	median	15	1	20.1704	0.1435	0.5764
NormalMaxForce	std	10	15	0.0130	0.0896	0.3681
PupilMove	std	10	13	1.7515	0.2428	0.0267
PupilMove	std	10	15	1.4574	0.2034	0.0717
PupilRadius	std	15	2	2.1452	0.0841	0.5001

results of our equivalence analysis. We represent the relative size of the offset by dividing ϵ by the average of the means.

The required ϵ values for all of the metrics we analyze are relatively small, compared to the rest of the data. However, a few of the t-tests (PupilMove, one of the AccelMag) are statistically significant: this would suggest that those particular sensor/window/offset combinations may not be the best choice for an eventual prediction system. The rest of the metrics behave well, however.

3.3.4. User Annoyance Prediction

In this section, we determine how accurately we can actually predict user annoyance.

To perform this analysis, we employ standard data mining techniques using labeled instances of data. We employ the Weka [54] data mining suite to assist in handling the data and building the models. In this analysis, we label any event where the user expressed annoyance as *True*, and instances where they did not as *False*.

We then create a large number of attributes on those labeled instances. To generate our attributes, we take data from 0–2 sec, 2–4 sec, 4–6 sec, 6–8 sec, 0–5 sec, 5–10 sec, 10–15 sec, 15–20 sec, 0–10 sec, and 10–20 sec, from before and from after each annoyance event. For each of those time periods, we find both the means and standard deviations. Additionally, we take the difference from before and after. This gives us a total of 60 attributes. We run this attribute generation method for each of our 7 sensor metrics.

If we were to simply use all of those attributes in a data mining algorithm, the most useful attributes would get lost in the noise of the rest of the attributes. Thus, to improve our eventual model's accuracy, we employ CFS [53], a feature selection algorithm, which reduces the number of attributes we end up using in the prediction model.

After feature selection, we then run the actual data mining algorithm. To build our model, we use logistic regression [125], which is a type of generalized linear model frequently used on binary labeled data. For all of our analyses, we use 10-fold cross-validation.

We generate a different model for each user. We also include a baseline set of data. This baseline was generated by using ZeroR, which is a naïve algorithm that always chooses the most common label (True or False). The results are presented in Figure 3.4.

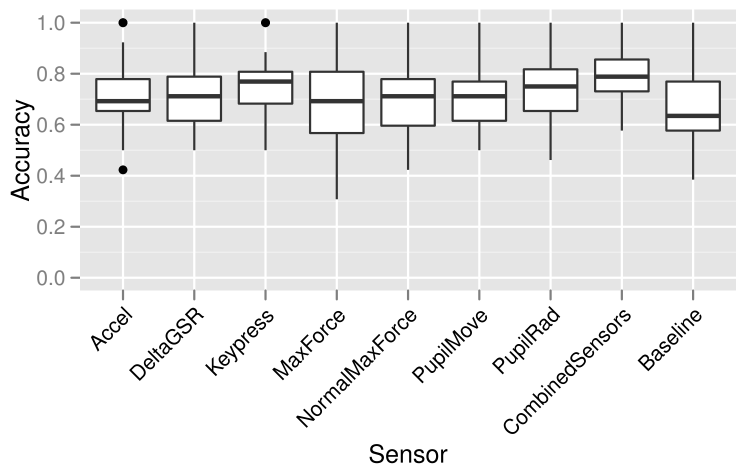


Figure 3.4. Model prediction accuracy using individualized user models. We include results for 7 individual sensors, a combined model which uses all 7 sensors, and a baseline prediction accuracy. The box's bottom and top are the 25th and 75th percentile, the middle line is the median, and the whiskers go to the highest and lowest points within 1.5 IQR of the higher and lower quartiles. Dots represent outliers beyond 1.5 IQR.

As the graph shows, all of the individual sensors have medians better than the ZeroR baseline case. However, when all of the sensors are combined in the model, the prediction accuracy is significantly better. We would like to note that some users did not indicate any annoyance, which gives ZeroR a slight advantage (since it is always correct for those users), but we did not discard the data; we assume that they could not tell a difference in performance.

We must also know how accurate the system is at individually predicting *True* and *False* instances, as a system which has a very high *True* prediction accuracy, but very low *False* prediction (or vice-versa) is not as useful. Using the combined sensors model, our system predicts *True* accurately 212/263=80.6% of the time, and *False* accurately 203/257=79.0% of the time (false positive=19.4%, false negative=21%).

3.3.5. Results Verification

In our original experiment, the users verbally notified us when they were annoyed. However, their verbal notification could alter the readings on our biometric sensors. To address this possibility, we run a small supplementary user study on five users to verify that we can detect when the user is annoyed without a verbal indication. The primary difference in this study is that we do not have the users notify us when they are annoyed; we simply have them play the game as the performance changes.

Since we already have determined which annoyances the user perceives as annoying and non-annoying, we only use the top 5 most annoying events, as reported by users in our original study (see Table 3.1), as well as 5 pseudo-annoyance events where there is no change in performance. We run each of those annoyance events two times. We increase the time of non-annoyance to 55–65 seconds, with 30 seconds of annoyance.

We then run a similar analysis to our original study, but instead of using the user-reported annoyances to label the annoyance events as *True* or *False*, we label all of the intentionally annoying events as *True*, and all of the pseudo annoyance events as *False*. The graphs in Figure 3.5 are the result of our prediction analysis. The baseline for this graph is exactly 50%: there were exactly 10 annoying events and 10 non-annoying events for each user, so ZeroR would always be correct 50% of the time.

As the graph shows, all sensors perform much better than the baseline, and again, the combined sensor model performs the best, with an average accuracy of 79%. This shows that our system works, regardless of whether there are any verbal user indications of annoyance or not.

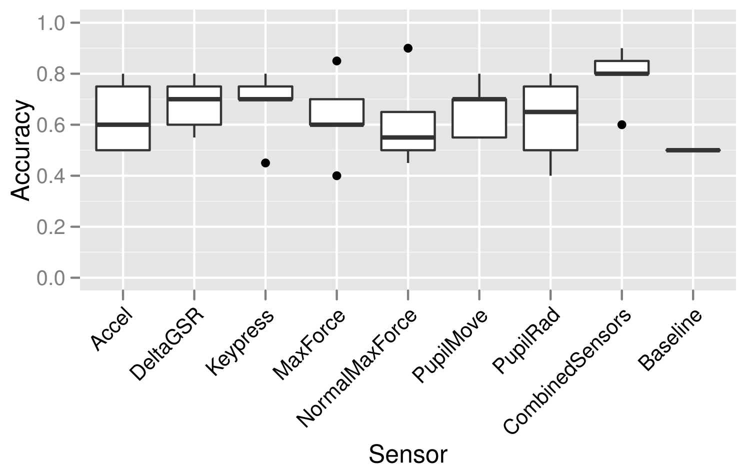


Figure 3.5. This boxplot is similar to the plot in Figure 3.4, but the users don't verbally indicate if they are annoyed. Data is labeled based on if the annoyances were typically annoying or not, which helps reduce noise caused by users accidentally mislabeling instances of data.

Using the combined sensors model, we are able to predict *True* instances 38/50=76% of the time, and *False* instances 41/50=82% of the time (false positive=24%, false negative=18%).

3.4. Conclusion

We have argued for measuring and understanding end-user satisfaction using physiological sensors, and supported our argument through a user study whose data allow us to correlate voiced satisfaction, biometric information, and intentionally-introduced power-savings events. We found that different power-savings techniques reduce user satisfaction to different degrees, and that saving a given amount of power using a combination of techniques is generally preferable to using a single technique alone.

We have also shown that it is possible to accurately measure changes in user satisfaction via various physiological metrics. When the performance is constant, biometric readings remain constant, while when performance varies, biometric readings also vary. This correlation can be used to predict user satisfaction from the metrics, and we showed how to do so with 80% accuracy through the use of per-user models developed through data-mining.

In a design environment where power and performance trade-offs directly impact user experience, understanding the relationship between user satisfaction and power-saving techniques is critical. This chapter highlights some of these effects, and indicates that biometric sensors are a powerful way to acquire feedback about the individual end-user without requiring any effort on their part.

CHAPTER 4

CAPED: Context-aware Personalized Display Brightness for Mobile Devices

4.1. Introduction

The display is the primary user interface in many devices across the computing spectrum. It is used as the primary output interface on traditional devices such as desktops and laptops. With the advent of the pervasive and ubiquitous computing era, devices such as smartphones, tablets, and smartwatches also use the display as the primary source of input to efficiently utilize their small size and form factor. This trend toward display-centric operation began with resistive and stylus-based PDAs and phones (such as the PalmPilot®), and reached a new level of popularity with modern smartphones and tablets. Thus, in many modern computing devices, the display quality is an important factor in determining the overall user experience.

The relative quality of a mobile display depends on many parameters such as its size, resolution, and brightness. Larger and higher-resolution screens have received widespread attention [79]. However, other display characteristics, brightness needs to be dynamically adjusted to suit the user in a given environment. For example, when checking emails in a dark room at night on their phone, a given user would tend to prefer a lower brightness, but when using their phone outside on a sunny day, that same user would tend to prefer a higher brightness. The choice of ideal brightness clearly depends on the ambient light in the surroundings. This led to the inclusion of ambient light sensors on many mobile devices. On most modern smartphones, the brightness of the display is set via the ambient light reported by these sensors [106].

However, this approach has two primary limitations. First, it ignores the difference in brightness preference from one user to another. Not all users are identical and one user might require a brighter screen for the same environment than another user. Second, the current approach does not include external context, apart from ambient light. We define *context* as any data which can be used to better explain a user's current state and their surrounding environment. We did a survey of existing research literature on perception [25, 56, 70] to discover existing knowledge on variance of brightness perception from one individual to another. We also conducted user studies to investigate the need for personalization as well as the need for using additional context information. Our studies presented in Section 4.2 clearly show that the simplistic, one-size-fits-all approach to controlling brightness of displays needs to be improved.

We propose a novel Context-Aware PErsonalized Display (`CAPED`) management system. The new system improves the existing state-of-the-art in two ways:

- We discover relevant context apart from ambient light which influences the preferred display brightness level, and include these in our new system.
- We develop an online learning-based approach which enables personalization to individual user preferences. Our algorithm is shown to theoretically and practically improve in accuracy over time as it adapts to user preferences. Further, we show that on average, we would never be worse than the current state-of-the-art technique which uses a fixed mapping for all users.

We validate our new system through a user study conducted on 10 users who were asked to use `CAPED`-enabled smartphones as their primary device for one week. Our user study demonstrated that the new system could more accurately predict the users' preferred screen brightness level than the existing state-of-the-art. The average improvement in absolute prediction error in `CAPED` is 41.9%. Users were also asked to rate the default and `CAPED` systems after a day of use. On a 5-point scale of satisfaction reported by the users, `CAPED` improves the satisfaction by 0.8 points, on average.

The rest of the chapter is organized as follows. We discuss the need for personalization and the need for including additional context information in Section 4.2. In Section 4.3, we describe our proposed system and its different components. We explain the experimental setup used in our user studies in Section 4.4. We present our results in Section 4.5, and we give a brief conclusion in Section 4.6.

4.2. Motivation

In this section, we present our motivation for the creation of an improved adaptive brightness model. The two primary pieces of motivation that we provide are the need for personalization and the need for the inclusion of additional contextual information in display brightness models.

4.2.1. Importance of Personalization

One of the most significant issues with current display brightness models is that they are generalized to the average user, and allow little room for personalized, per-user settings. These models assume that all users have the same screen brightness requirements, and so screen brightness is controlled accordingly. We don't believe that this is a sufficient approach to display brightness control, and we present some supporting evidence here.

If you remember back to Section 2.1, Figure 2.1 showed aggregate readability for people from a wide variety of ages. Because aging causes such a significant degradation in the required contrast for a given RVP level, this suggests that a generalized model can be improved upon with a more personalized model, as age is not considered at all in generalized brightness models. Furthermore, it is important to note that these graphs contain data aggregated across a number of users for a given age. Simply knowing one particular user's age does not mean that that user's individual RVP curve can accurately be determined; there is a large amount of additional variability even between users of a similar group.

For example, visual acuity can vary from user to user. Visual acuity is a measurement of the clarity of a person's visual system, and is dependent on a variety of factors, including the quality of the focused image on the retina, the proper functioning of the retina itself, and the ability of the nervous system and brain to transmit and interpret the visual data [26]. A user with poor visual acuity will have worse RVP measurements than a user with good visual acuity in the same ambient light and contrast range.

There is even variability among two users with similar visual acuity. Contrast sensitivity is a measurement of how well a user can discern contrast in a given scene [56]. Even a user with perfect visual acuity may struggle with contrast discernment tasks. Since contrast sensitivity is directly related to the contrast curves in the RVP metric, this further complicates the feasibility of a generalized brightness model.

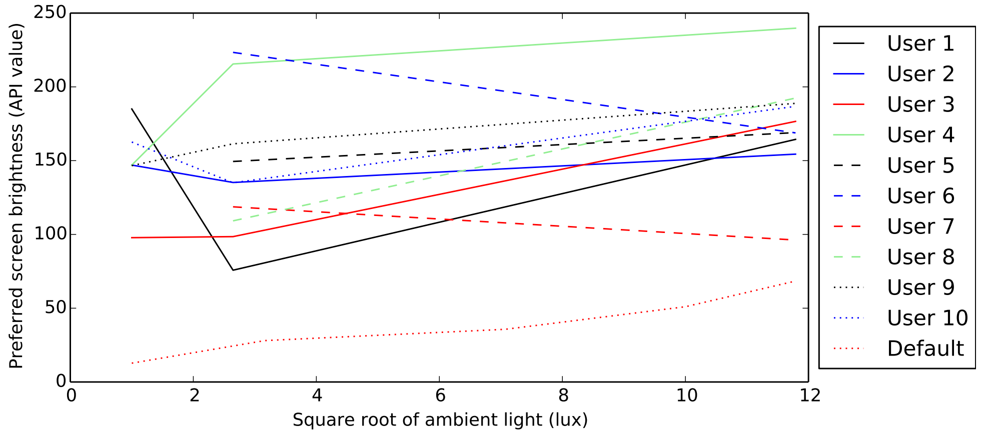
To further experimentally motivate our work, we run an initial, controlled user study. A series of users were seated with a Google® Nexus™ 4 smartphone. We then asked users to indicate their preferred brightness levels while we artificially manipulated the surrounding ambient light levels and on-screen images. Although RVP is solely based on textual readability, our motivating study looks at brightness requirements for both images and textual content.

The results from this study are shown in Figure 4.1. This data contains 4 dimensions (the user, the displayed image, the ambient light level, and the user's brightness preference). To show the underlying trends in this data, in Figure 4.1a we average the results across different images, while in Figure 4.1b, we average our data across users.

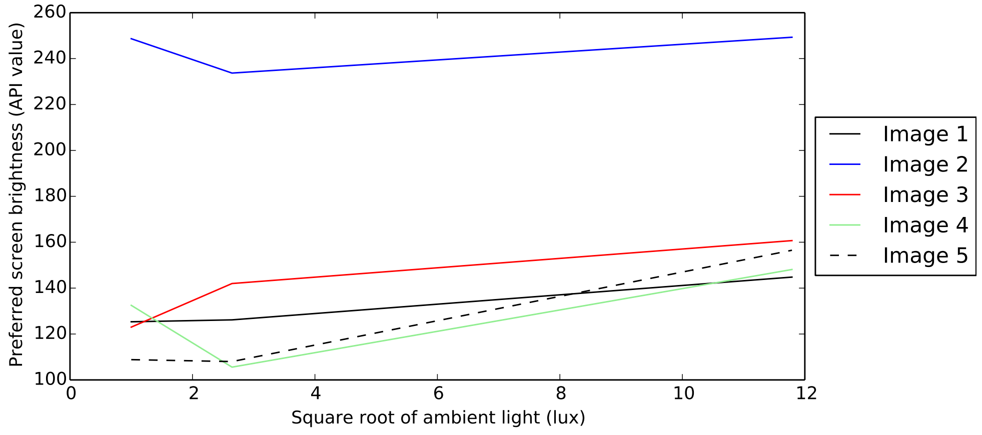
As Figure 4.1a shows, not only do users' display preferences differ from one another significantly, but they also differ from the default brightness model on the device. There is a general positive correlation between brightness and ambient light, but some of the users do exhibit either negative or little correlation of their preferred brightnesses with ambient light.

Figure 4.1b suggests that required screen brightness varies not just for text, but also for images. It's interesting to note that the actual on-screen image impacts the preferred brightnesses; some on-screen content demands higher or lower brightness levels than others for certain users.

Given the fantastic complexity of any individual user, we believe that a generalized model of user brightness preferences is infeasible at best, and impossible at worst. Because of this, we propose that to accurately predict screen brightness levels, we must use a personalized, per-user brightness model. Furthermore, we also believe that there are



(a)



(b)

Figure 4.1. Results of initial study which analyzed users' brightness preferences for a selection of on-screen images and ambient light levels. Figure 4.1a suggests that user brightness preferences differ from one another, and from the manufacturer's default brightness model. Figure 4.1b suggests that image content significantly impacts brightness preferences.

external factors besides the visual system and readability which may significantly impact an individual's brightness requirements at any given point in time. We outline these external factors in Section 4.2.2.

4.2.2. Contextual Data Inclusion

In Section 4.1, we defined context as any data which can be used to explain a user's current state and their environment. Currently, baseline models for screen brightness requirements are solely based off of contextual data regarding ambient light. Ambient light contextual data allows the model to attempt to provide a more constant level of readability to the user. However, we believe that there are other contexts that are important in accurately predicting preferred screen brightness. Here, we outline the contextual data that we include in our proposed system, and provide some intuition behind the decision of using the context in our model. This is not meant to be an exhaustive list of all possible contextual data, but as a reasonable starting point for a more contextually-aware system.

Circadian rhythm is a 24-hour cycle that many biological processes are based around. Excess artificial light is known to have the potential to disrupt the regularity of this cycle [37]. This can make it difficult to fall asleep, stay asleep, wake up in the morning, or function at full energy throughout the day. Because of this, we suspect that users may have differing brightness requirements depending on where the sun is in the sky. We don't believe that this context should be used in a generalized model, because people have their own natural tendency toward being more active at night or during the day (night owls vs early birds) [76].

We also believe that the duration a display is active can be a good predictor of brightness requirements. We propose using device on-screen time, which is a measure of how long it has been since the user turned their screen on, as a useful piece of contextual data. Since phones provide their own light source, the human eye adjusts to this source of light altering its physiological characteristics [3]. We intuitively expect that as users become acclimated to the phone's brightness, they may have varying brightness requirements.

Accelerometer data is another potentially useful piece of contextual data. The more that a phone (and the user) moves around, the harder it becomes to see the screen, reducing RVP at a given screen brightness level [92].

A similar context, activity characterization, may also improve brightness predictions. Activity characterization uses a combination of on-device sensors to predict what a user is currently doing (e.g., still, walking, biking, driving, or tilting the screen). Because visual acuity or the amount of constant attention that can be given to the display may differ between these activities, we believe this context to be potentially useful.

Battery level is another interesting piece of contextual data. We anecdotally observed that some users adjust their screen brightness depending on how much battery life is remaining. If the remaining battery level is low, the tendency is to dim the screen to extend the remaining on-screen time, which suggests that this context may have some predictive power.

Finally, we include location data as an input context in our model. We believe that the user's current location (work vs home, etc.) may impact a user's display brightness requirements.

In Section 4.2.1, we noted that on-screen content can be a strong predictor of required display brightness. However, for the reasons specified in Section 4.3.3, `CAPED` is implemented as an application-level piece of software rather than integrated at the platform level. Because of the UI lag that including display content as a context causes, we don't include display content contextual data for this study. We may, however, explore the inclusion of screen content at the platform level in the future.

4.3. System Description

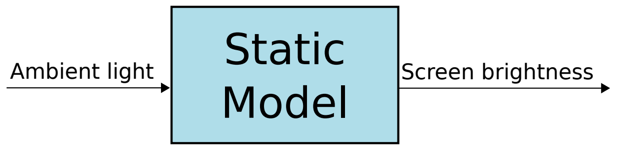
As outlined in the previous section, there are a number of shortcomings that current adaptive screen brightness systems currently have. The primary deficiencies we identified are that current adaptive screen brightness systems are one-size-fits-all, and that providing additional contextual data can improve adaptive screen brightness prediction accuracy. In this section, we describe `CAPED`, our proposed system for addressing these deficiencies.

4.3.1. Proposed Model Description

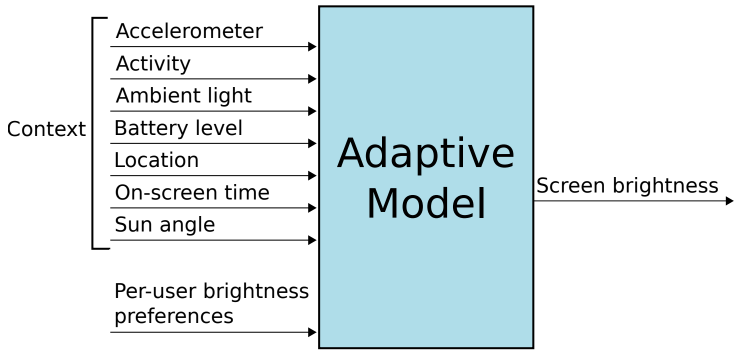
To describe our model for predicting user satisfaction, we first begin with the manufacturer default ambient brightness model, and gradually modify that model to incorporate personalization and context awareness in the model. As shown visually in Figure 4.2a, the sole input to the default prediction model is ambient light information. A simple, fixed function is then applied to the ambient light values, and a predicted value for that ambient light level is generated.

This model is completely static, which means that no personalization can be performed. To address this, we allow users to directly provide our model with preferred brightness levels. With each new brightness preference indication, the adaptive model is updated to better match the user preference.

It's important to note that one of our top priorities in this system was to improve the adaptive brightness model, but only if we can do so without negatively impacting the user



(a) Default model: The only input to this model is ambient light, and the model is static.



(b) `CAPED`: we add additional inputs to the system, and also allow users to provide their preferred brightness levels at a given context. `CAPED` is updated as additional input is provided.

Figure 4.2. Baseline and proposed adaptive brightness management.

experience. Thus, we do not want to interrupt users or disturb their device session in any way to get brightness preference information. Instead, users make their brightness indications only when they are dissatisfied with the brightness state. Their indications are easily made via a button in the notification area, which displays a slider that allows them to select a preferred brightness level.

The second aspect of the default model we aim to fix with `CAPED` is that the only context used as an input to the model is ambient light. In Section 4.2.2, we presented evidence that ambient light is not the only important predictor of preferred screen brightness. To allow additional contexts as inputs to our model, `CAPED` enables an arbitrary number of contexts as input features, targeting screen brightness as our predicted output. This proposed system is described in Figure 4.2b.

Thus far, we have been intentionally vague about the adaptive model, which learns user brightness preferences with some number of contexts as input. There are any number of possible models which can be used to predict an output given some set of inputs. Something as simple as a linear regression may be able to accurately predict an output, or it may require a more advanced general-purpose model such as support vector machines, or even a non-linear model like decision trees. Furthermore, an adaptive model doesn't even need to be a general-purpose model; if some amount of domain knowledge is available about the output, a hand-crafted model may also be effective. Because we don't know exactly what model will best suit a given user, we don't want to pre-select a single adaptive model in our system. Instead, we employ the use of several simultaneous learning models, each of which provides their own outputs. This allows us to eventually prefer predictions from the most accurate sub-models over those from the least accurate ones. We more clearly define this system in the following section.

4.3.2. Online Model Composition

Because we plan to use a number of simultaneous learning models for predicting user brightness preferences from contextual data, we need a meta-algorithm to do online selection of models to minimize prediction error. Some models may be better predictors for certain users and in some context. Automatically inferring which model to use for prediction is a classical online learning problem. Blum provides a detailed survey on online learning, describing different techniques to select prediction models adaptively [15]. One of the classes of online algorithms is the weighted majority algorithm [86]. This algorithm is a binary classification algorithm which contains a pool of individual binary classifiers, each of which classifies data independently. In addition to these independent classifications, each sub-algorithm contains a weight. As training data is introduced to the sub-algorithms, these weights are either increased or decreased in value by some function depending on if the classification was made correctly or not. To classify a new piece of data, the weighted majority algorithm compares the weighted value of the binary "0" predictions to that of the binary "1" predictions from each of the sub-algorithms. The prediction with the higher weight is the selected prediction from the model. This algorithm is useful to `CAPED` because the weighted majority algorithm gives upper bounds on the number of incorrect predictions that the meta-algorithm contains.

However, the weighted majority algorithm [86] only predicts binary data; screen brightness is continuous data, and so we need to modify the weighted majority algorithm to work with a non-binary output. We use the continuous variant of the weighted majority algorithm proposed by Vovk [122]. Instead of using the weighted summation of each sub-model's predictions, each prediction is instead a weighted combination of the sub-models, where $\hat{p}$ is an individual prediction, N is the number of experts, w is a weight, and f is an individual sub-model's prediction, as shown in Equation (4.1). The output from this formula is continuous rather than binary.

$$\hat{p}_t = \frac{\sum_{i=1}^{N} w_{i,t-1} f_{i,t}}{\sum_{i=1}^{N} w_{i,t}} \tag{4.1}$$

As new training data is added to the online meta-algorithm, the weights on each sub-model are updated depending on how close their prediction was to the result, where y is the actual outcome and l is some loss function:

$$w_{i,t} = w_{i,t-1} e^{-\eta l(f_{i,t}, y_t)} \tag{4.2}$$

The most important piece of this algorithm for `CAPED` is the upper bounds on accuracy that it provides. Because we don't know in advance which algorithms will most accurately predict user state, we wish to allow this system to gravitate to the most accurate classifiers. Over time, the number of errors in the meta-algorithm is known to converge to the number of errors in the best sub-algorithm. See [122] for theoretical proof of convergence. Thus, given a sufficiently large number of predictions, the meta-algorithm will not do worse than any of the sub-models. This is represented by Equation (4.3), where $\hat{L}$ represents the error from the online model composition, and J is the pool of experts.

$$\hat{L}_n \leq \min_{i \in J} L_{i,n} + \sqrt{\frac{\ln(N)}{t}} \tag{4.3}$$

In our specific implementation, since we include the manufacturer's default brightness model as one of the sub-models, this implies that on an average after adequate training points, we will not perform worse than the default brightness model. This is important in situations where none of the other sub-models are able to accurately capture the user's preferences, or if the user gives noisy inputs to the system.

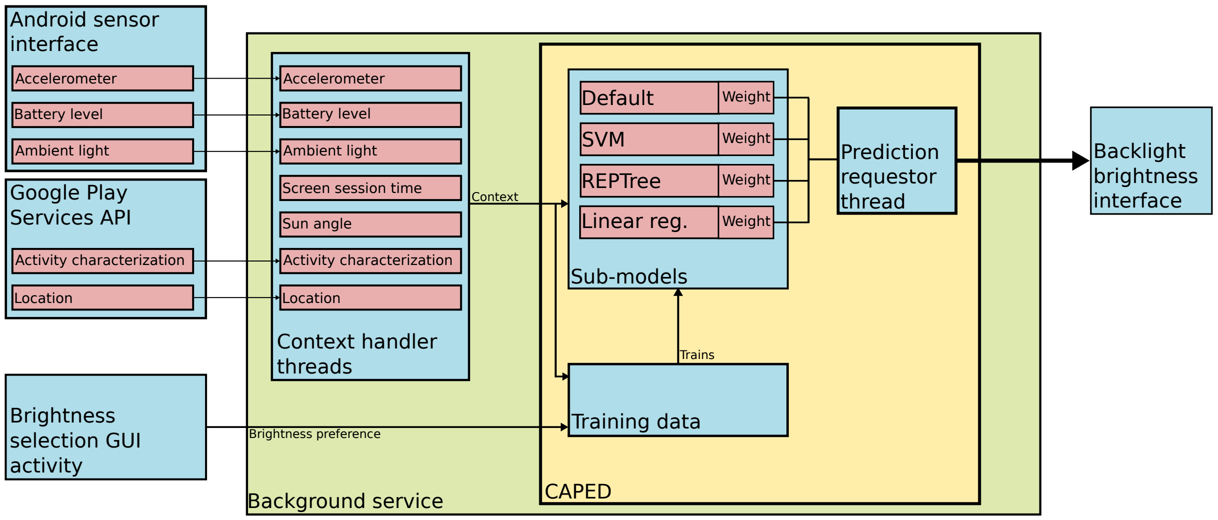


Figure 4.3. Integration of CAPED with the Android operating system.

Another interesting side-effect of this meta-algorithm is that the most heavily weighted sub-algorithms weights can change throughout the system's lifetime. Because a heterogeneous set of sub-models are used in this system, it may happen that some algorithms work sufficiently well with a small training set, while others work better with a large training set. This allows the meta-algorithm to prefer sub-models which have the best prediction accuracy, even if the most accurate models change over time.

4.3.3. CAPED System Architecture

In this section, we describe the specifics on how CAPED is integrated with the Android Nexus 4's operating system. A summary of this architecture is shown in Figure 4.3.

CAPED is programmed entirely in Java, and exists as a user application which manifests itself as a conglomeration of a number of background services and user-facing activities. CAPED's contextual data gathering is accomplished via standard Android userspace API calls. We described the motivation for including the various contextual data in Section 4.2.2;

we now give specifics as to how these contexts are collected, and what data we specifically extract from them. All of the contextual information that we gather are from either on-board sensors, third-party Google APIs, or calculated programmatically. The accelerometer, ambient light, and battery contexts are all sourced from on-device sensors via the standard Android application API. The activity characterization and raw location data is obtained from Google's location and activity characterization APIs. We perform some additional calculations to cluster the locations into clusters with a maximum radius of 0.5km. Finally, on-screen time and the sun's angle are calculated using the device's clock. Each of these contexts are calculated in their own individual thread, which allows us to configure update rates individually.

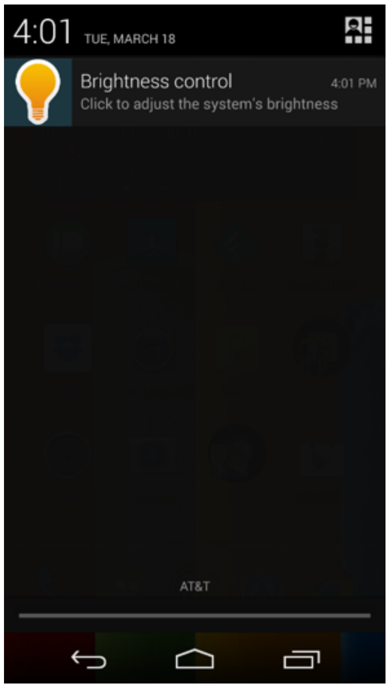
`CAPED`'s meta-algorithm is implemented as follows. The meta-algorithm contains a pool of sub-models, each which has a weight associated with it. When a prediction is run, then all of the current contextual data is packaged into a data structure, passed to all of the sub-models individually, and each sub-model makes a prediction using that contextual data. The predictions from all of the sub-models are gathered, multiplied by their weights, and that result is combined into one final meta-output.

We generate display brightness predictions at a rate of 1Hz. To control the display brightness, we use API calls which manually set the device's screen state using the meta-algorithm's meta. The device's default adaptive brightness system is disabled when `CAPED` is active.

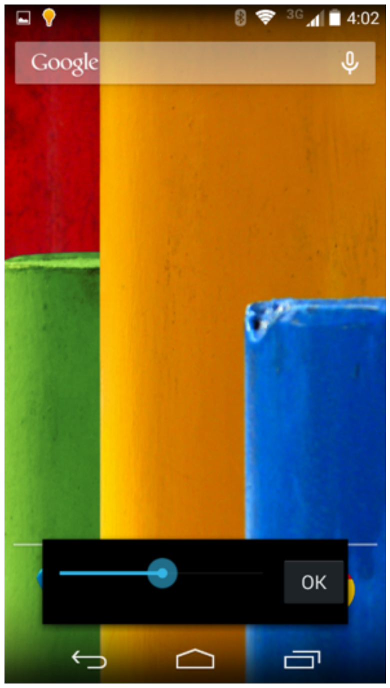
To add training data to `CAPED`, a button is displayed in the device's notification area, as shown in Figure 4.4a. Selecting this button displays a dialog box which allows the user to select their preferred brightness level (Figure 4.4b). When the user selects their preferred brightness level, a series of events happen. First of all, that brightness indication is packaged up along with the current contextual data. The sub-models then each run an individual prediction (without using the new training data yet), and depending on how close their prediction was to the actual preferred output, the sub-models' weights are updated accordingly. Finally, the sub-models themselves are updated with the new training data. None of the general purpose models we use are *updatable* models; this means that with each new piece of training data, the adaptive sub-models are rebuilt from scratch using the cumulative set of training data. Using updatable models could optimize the amount of time that it takes to add a new piece of training data, but wouldn't have any impact on the model accuracy. Even on larger training sets (≥ 1000 instances), generating a sub-model takes less than a second, and this only occurs when the user makes a new brightness indication, so we don't pursue updatable models in this work.

The general purpose sub-models are provided by the Java-based Weka [54] machine learning library. Weka is shipped as a Java-based machine learning solution with GUI support; this GUI support had to be stripped out to be compatible with Android. We also include the manufacturer's default model as one of the sub-models. This sub-model is completely static; it doesn't change as new training data is added.

We developed `CAPED` as a user application for the sake of improved portability, rapid development, and ease of deployment (allowing us to eventually distribute the application to the public). However, there are some advantages that could be provided by implementing this system at the platform level. First of all, a platform-level implementation would allow for direct control of the screen brightness. Manually controlling the brightness via API calls means that the operating system ultimately controls the transition between brightness



(a) Navigation menu button to bring up `CAPED`'s interface.



(b) `CAPED` brightness setting interface, appears once button in Figure 4.4a is pressed.

Figure 4.4. `CAPED` user interface

levels. A platform-level implementation of `CAPED` would allow for fine-grained control over the brightness transitions, but these transitions are out of scope for this work. Secondly, accessing the display's image content at the application level is a computationally intensive process which introduces severe UI lag into the system when display content is analyzed at the rates that our system requires. Platform-level access could make the inclusion of screen content as a contextual input feasible by reducing the CPU and bus bandwidth overhead of the calculations.

4.4. Experimental Setup

In this section, we describe the experiments we conduct to measure `CAPED`'s effectiveness. We perform user studies on a set of 10 smartphone owners. Participants were gathered via fliers advertising the study.

Each of the users was provided with an Android Google Nexus 4 smartphone to use as their primary mobile device for the period of one week. We used a standard device for all of the studies because display brightness models vary between devices, and this allows us to better understand the user preferences compared to the default model. Users' existing cell phone plan was used to provide data access on the Nexus 4 device. The users were instructed to use this device as if it was their own; they were allowed to install any applications on the phone that they would typically use, and were told to customize any additional settings on the device as they see fit. `CAPED` is installed on each of the devices.

The number of users in this study was limited to 10 because the study lasts a number of days per user, and because we provided the equipment for the users. We wanted to have all users using the same device model, not because of any technological limitation (`CAPED` has been successfully run on many other models), but because we wanted to analyze how different users interact with the same display interface. Our sample size of 10, although not large enough to precisely describe every way that a person could possibly interface with `CAPED`, is certainly large enough to give results that show some variety in users, while repeatedly demonstrating the effectiveness of `CAPED`.

While they were using the device, the users were instructed to indicate their preferred brightness levels whenever they were dissatisfied with the display's brightness. Each time the user made a brightness selection, that preference, along with the current contextual data, is added to CAPED's training set.

CAPED uses four sub-models: the default Nexus 4 brightness model, an SVM regression model, a linear regression model, and a decision tree model. The default model is initially given a weight of 1, while the adaptive models are given weights of 0.0001. We weight the models this way because the default model has been pre-calibrated to suit the average user, and we expect that it will have the best accuracy without any training data provided. If the generic machine learning models begin to predict brightnesses more accurately than the default model, they will become more highly weighted than the default model, and those predictions will contribute more to the model's output.

To better gauge how users perceive our brightness algorithm, we split our study into a few different phases. During the first three days of the study, CAPED is used to control the backlight brightness, and the users' brightness indications are used to train the model; this is how CAPED would function in a real system. On the final four days, two of the days use our brightness prediction model, and two of the days solely use the manufacturer's default brightness model, in a randomized order. At the end of each of those 4 days, users are asked to rate their satisfaction with the screen brightness during the previous 24 hours. This allows us to subjectively gauge how satisfied users were with CAPED.

To get more information about the user's surrounding context, we also log contextual data two times per minute. This data collection is not at all tied to the user indications.

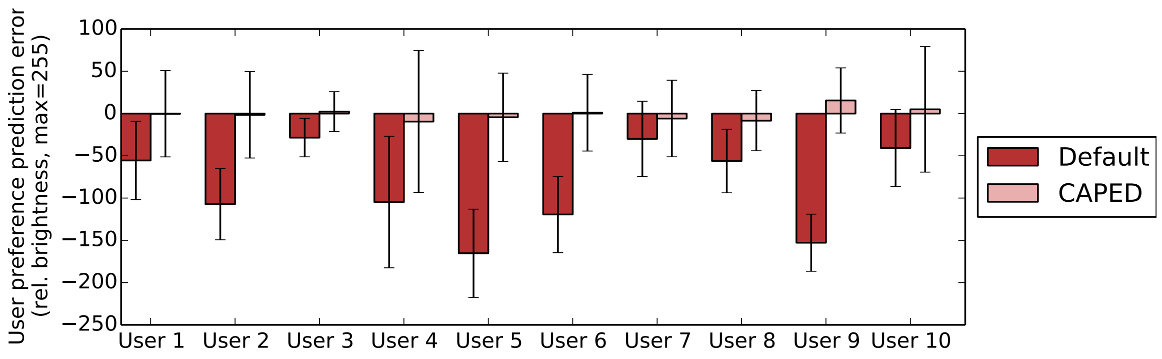


Figure 4.5. Comparison of prediction model accuracies compared to the actual user-indicated brightness preferences. The mean error is presented in units of relative screen brightness, which is used by the Android brightness settings API. Bars represent one standard deviation. Minimum brightness=0, maximum brightness=255.

4.5. Results

In this section, we present our results for describing the effectiveness of `CAPED`, and compare it to existing screen brightness models.

4.5.1. Prediction Accuracy Analysis

In this section, we describe the accuracy of `CAPED`, compared to the accuracy of the default model. During the user study, each user was asked to adjust their preferred brightness settings when they were dissatisfied with the current brightness. For each of these preference indications (and before using this data to update `CAPED`'s prediction model), we collected the predicted brightness value for the given contexts from the default and from `CAPED`. From these predictions, we calculated the error for both models, for each indication. These errors were then averaged and are presented in Figure 4.5.

Across all users, the average error of the default model is -86.03, while the average error of `CAPED` is -0.05. The mean reduction in absolute prediction error, averaged from each user's errors, is 41.9%. Although Figure 4.5 shows that the default model typically does underpredict the user's brightness requirements, it does not mean that simply increasing the default model's brightness curve would cause the system to perform as accurately as `CAPED`. It isn't the common case, but there are many instances where `CAPED` predicts a brightness lower than the default model, while remaining more accurate than the default model.

It is important to also consider the effectiveness of using a meta-algorithm with multiple sub-models, rather than just using a single adaptive model. For a majority of the users, the progression of the weights for each sub-model tends to follow the same general pattern. The default classifier always begins with the highest weight, since we manually initialize the sub-models as such. Since the user requirements typically significantly differ from the default classifier, as the users continue to train the meta-algorithm, one or both of either the SVM regression or the linear regression models quickly becomes more heavily weighted than the default classifier. Furthermore, the decision tree model was overall the most accurate classifier, but only with a relatively high number of indications (at least 15–20 or so). Hence, the decision tree's weight typically remains relatively low until a sufficient amount of training data is provided. This weight trend isn't identical for all users, as some users behaved more linearly than others, but was the most common case.

This progression of the sub-model weights shows the merit of using multiple sub-models, rather than only one. The most accurate sub-model tends to change over time; the linear models typically perform better with a small amount of training data, while the more complex models tend to perform better with larger data sets. Thus, the best model is actually a combination of the subclassifiers. Furthermore, it is common (in 70% of cases) for multiple classifiers to contribute significant weights to the prediction at the end of the week-long study, rather than completely converging to a single model. This furthers the case for using multiple sub-models.

4.5.2. Impact of Contextual Data Inclusion

In this section, we describe the impact that the inclusion of additional contextual data has upon the overall prediction accuracy. To accomplish this, we run an offline analysis of the user brightness preference and indication data. Using an exhaustive subset analysis, we determine the subset of contexts for each user which maximizes that user's brightness value prediction accuracy. We then compare the absolute error of running predictions with this optimal subset to the error of running predictions using ambient light as the sole contextual feature. This method allows us to see how much the accuracy can be improved via additional contextual data, as well as determine which contexts have the most predictive power. Using this analysis, we find that using the optimal subset of contexts has a 14.5% lower error rate than using just ambient light as the sole input.

In Figure 4.6, we present how frequently each given context is included in the various users' optimal subsets. To calculate this, we simply take each user's optimal context subset and count how many times each context is included in the optimal user models. Expectedly, ambient light is one of the most important predictive contexts, but additionally both the current location and the position of the sun are frequently included in the optimal subset model. The motion of the device seems to have a very low level of predictive power.

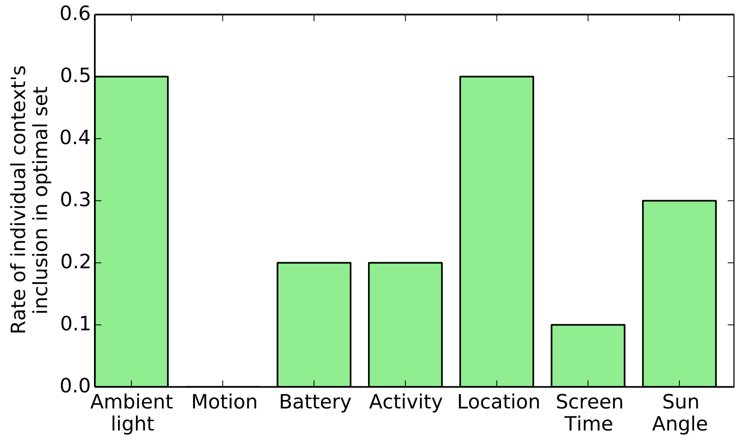


Figure 4.6. The rate at which each context was included as part of a user's optimal subset of context features. The optimal subset was determined via an exhaustive subset search.

4.5.3. User Rating Comparison

For the final four days of the study, two of the days control the display using the default mode, while two of the days use `CAPED`. Specifically, at the start of each day, we select the model for that day randomly (the users are unaware of this change). The random selection continues until one of the schemes is selected twice, after which point we continue with the other method for the remainder of the 4 days. At the conclusion of each day, the system prompts the user to rate their satisfaction with the system's brightness between 1 and 5, with 5 being the most satisfied. The average ratings for each model and each user are presented in Figure 4.7.

As the figure shows, `CAPED` universally either doesn't impact ratings, or improves them. Also interestingly, in general the users with the largest increase in rating tend to be the users with the largest improvement in mean prediction error. This suggests that the prediction error is, indeed, a strong contributor to satisfaction with the screen brightness.

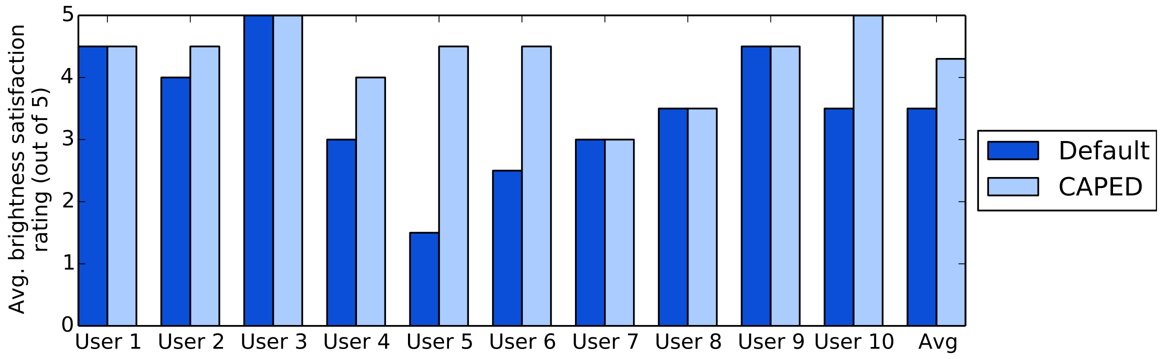


Figure 4.7. Comparison of user-indicated model satisfaction ratings over a 24-hr period. Users use one of the two models for two days each, for a total of 4 ratings. Mean ratings on a 1–5 scale are presented here.

On average, `CAPED` improves the default scheme's subjective user satisfaction rating from 3.4 to 4.2, or a 0.8-point (23.5%) improvement on a 5-point scale.

4.5.4. System Overhead

The primary design goal for `CAPED` was to accurately predict the preferred screen brightness; we intentionally allow `CAPED` to exceed the default model's brightness levels if it is closer to the user's preferences. As Figure 4.5 suggests, `CAPED` tends to exceed the default model's brightness levels. Additionally, as one would expect, the contextual data collection and prediction systems of `CAPED` have an additional impact on the device's power consumption. Because increased power consumption is a concern on any battery-constrained device, in this section we examine `CAPED`'s impact on power.

During the user studies, we periodically logged the device's contextual data. Even though only one of either `CAPED` or the default model can be active at a time, we can use the contextual data to "replay" what the screen brightnesses would be over the same periods of time. We additionally generate a linear model which relates the screen's brightness to the amount of current being discharged from the battery. We create this model by gathering amperage data from the Nexus 4's internal current sensor (part of the device's power management IC) at a variety of backlight brightness levels. This model, along with the brightness data of the two schemes, allows us to determine the difference in power consumption due to screen brightness between the two schemes over time. From this data, we then integrate the current flow to find the total difference in mAh (milliamp hours) between the models for each user per day, using their actual usage trends. Finally, because the Nexus 4's battery has a known capacity of 2100mAh, we then calculate the difference in the percentage of the total battery capacity that is consumed by the screen per day.

`CAPED` slightly increases the screen's power consumption for all users. Across our participants, the smallest increase in daily battery capacity usage is 0.5%, while the largest increase in daily battery capacity usage is 5.5%. On average, `CAPED` will cause the screen to consume an additional 2.1% of the total battery capacity per day due to increased screen brightness. This is a small increase in power compared to the display's overall power consumption [117]. On average, each user used the Nexus 4 device for 55 minutes each day. Although there is a general correlation between on-screen time duration and increased battery consumption, it interestingly is still heavily dependent on the individual users' brightness preferences and contextual data. For example, one user had a daily on-screen average of 149 minutes with a total daily battery consumption increase of 3.5%, while another user had a daily on-screen average of 49 minutes, with a 5.5% battery consumption increase.

Another source of power overhead is with regard to the prediction system itself. We compared the system's idle current draw while the screen was active and set to minimum brightness, with and without CAPED installed. Without CAPED installed, we found the Nexus 4's average idle current draw to be 212mA. Collecting the same data, but with CAPED running predictions and collecting contextual data, the average current draw is 280mA. However, the majority of this increase is due to the activity characterization collector; with this piece of contextual data disabled, CAPED only draws 228mA. It's important to note that CAPED only increases power consumption while the display is actually on; the majority of the time, the devices weren't actively used. Because of this, the amount of total daily battery capacity consumed by CAPED is 2.83% on average with activity characterization, and only 0.67% with activity characterization disabled.

There is also a system overhead that applies to the users themselves; a training system which requires constant interaction but improves the prediction accuracy could still reduce the user's overall satisfaction with the system. On average, users adjusted the display brightness 4.4 times per day when CAPED was active. This training set size, which is small in terms of typical data mining applications, suggests that is possible to get accurate brightness predictions without requiring a high number of user indications. In addition, we believe that the number of user inputs will drop down even further once CAPED has sufficiently learned the user requirements.

4.6. Conclusion

In this chapter, we proposed CAPED, a system which enables personalized, context-aware screen brightness predictions. We outlined the necessity of an intelligent brightness control model, and implemented our system as a userspace application. We then ran user studies to gauge the accuracy of our system's predictions. Our results showed that the manufacturer's default model for predicting user screen brightness is insufficient for many users, and showed that we can increase the mean absolute prediction accuracy by 41.9%, and improve the user's average satisfaction with the display brightness levels by 0.8 points on a 5-point scale.

As the focus of personal computing devices continues its shift of focus from raw power and specs to device usability, we believe that the user experience will be one of the most important ways for device manufacturers to differentiate themselves. That shift in manufacturer focus reflects a shift in user focus as well — a significant number of users want devices to suit them, rather than having to micromanage every aspect of their electronic devices. This increased need for automatic preference configuration and prediction lends itself well to on-device online learning systems.

We also believe that there are many other mobile subsystems which could benefit from online learning.

CHAPTER 5

Optimizing Mobile Display Brightness by Leveraging Human Visual Perception

5.1. Introduction

In modern mobile devices, the touchscreen display has taken front and center as the most important interface that users interact with. Not only does the touchscreen display act as the primary means of output to the user, but with the absence of hardware keyboards and buttons on many mobile devices, it is now the primary means of input as well. With this elevated station that touchscreen displays have come to enjoy in smartphones and tablets, more and more focus is being placed on the design and features of the display. The trend of mobile displays is increasingly toward larger, brighter, higher-resolution displays [79], and this trend is undoubtedly a sight for sore eyes everywhere.

However, this trend toward larger and brighter displays does not come without cost; although many manufacturers tote the energy efficiency of their displays, Chen et al. [21] find that generational improvements in OLED displays account for minor improvements in power consumption. They find that after two Samsung display design iterations, normalized full-brightness per-pixel power efficiency only improves by 5.7%; this is reduced even further when the current trend toward increasingly larger screens is considered. Furthermore, they find that the most significant contributor to power consumption is the size of the screen itself. With larger devices, although the battery tends to be bigger, the screen is also larger, and considering the lack of significant generational improvements in power consumption, it's expected that display power will continue to be a significant consumer of power in mobile devices.

Because the display consumes a significant amount of power, adaptive brightness systems are one way that display manufacturers attempt to reduce the display's power envelope while improving user satisfaction with the display's brightness. Adaptive brightness systems control a display's brightness via ambient light sensors which give an indication of incident environmental light on the screen. These adaptive brightness systems can significantly reduce the power usage compared to a display which is constantly set to a high brightness level. Also, adaptive brightness systems have the advantage of causing less eye strain by avoiding setting the brightness too high when in a dim lighting environment [94].

In this work, we wish to gauge how effective these adaptive brightness systems actually are at satisfying users' brightness requirements. To do this, we design a user study which allows us to gather subjective display brightness satisfaction ratings and readability data for a series of users at a selection of ambient light and screen brightness levels. From these results, we find that the default automatic brightness system that comes shipped with the device isn't well-tailored to the group of users we studied.

Furthermore, we find that subjective user ratings are closely correlated with an objective readability metric, especially in brighter lighting environments. This strong, direct correlation is interesting because it suggests that by using this relationship, it's possible to understand how much of an impact selecting a given brightness level will have on a user, without needing to directly gather subjective ratings from the user.

We also study the impact that various brightness throttling schemes have on users. System power may need to be reduced below the standard power envelope for many reasons. For example, one common reason for reducing display brightness is low-battery situations. When the battery falls under a critical threshold, many smartphones reduce the display's brightness to help extend the device's remaining on-screen time. Additionally, there are systems such as *system-level dynamic thermal management* [77], which reduce system power consumption in an attempt to reduce the system's total heat levels. Any OS subsystem can cause power throttling events, but the underlying issue with these systems is that they do not consider the impact that these power savings decisions have on the user.

To explore this further, we conduct another study which collects data on what lighting environments are commonly encountered throughout the day by a typical user. Using these results along with the previous readability and ratings data, we directly analyze how much power can be saved with a specified degradation in readability. We also compare a user-agnostic system which degrades the display by a constant brightness level or constant brightness percentage, to our proposed user-aware system which degrades the display's brightness more in brighter environments. Any backlight dimming power savings decision will impact users to some degree, but by considering how these decisions affect the user, it's possible to reduce the average impact over a period of time. Using our user-aware brightness throttling scheme can achieve an 8% average system power reduction while only degrading average readability by 3.2%; this is 21.5% less of a degradation than the next best user-agnostic method. Even more importantly, this method allows us to directly understand how the power decisions will affect users, which makes these power decisions more well-informed than arbitrarily throttling system power.

In summary, we make the following primary contributions:

- We find a correlation between subjective user ratings and an objective readability metric
  - We run user studies which gather subjective ratings and visual discernment task data across multiple lighting environments
  - We present analysis which shows a strong correlation between objective visual discernment and subjective ratings
  - We perform the first analysis on the impact of adaptive screen brightness mechanisms on readability
- We create a model for screen brightness which maximizes time-averaged screen readability
  - We compare existing adaptive brightness systems to a properly calibrated one
  - We create a power throttling model which maximizes time-averaged readability for a given power reduction
  - We compare our system's time-averaged readability against other power throttling models

In Section 5.2, we briefly outline introduce our experimental device's brightness and power characteristics. Section 5.3 outlines the user studies that we run to gather data for our analysis. We present our results in Section 5.4, and conclude in Section 5.5.

5.2. Device Characteristics Validation

In this section, we briefly introduce the power and brightness characteristics of our targeted smartphone device. We use the Google® Nexus™ 4 smartphone, which contains a 4.7”, 320ppi IPS LCD display [95]. LCD displays are more common and more mature than the newer OLED displays, but the two behave similarly from an optical standpoint.

Most modern mobile displays have brightness controls which can be accessed programmatically by the operating system. However, these brightness values as reported by the OS aren't guaranteed to be directly related to the display's actual brightness. Because of this, we use an external light sensor to verify that the display brightness levels indicated by the operating system are linear with the screen's actual brightness (Figure 5.1), and find that they are linearly correlated. It is interesting to note that the granularity of the measured screen brightness is lower at the higher screen brightness levels. This is likely a design decision made by the display manufacturer; the human visual system's sensitivity to change in stimuli is reduced at higher brightnesses, making it unnecessary to implement all of the brightness levels on the display [119].

We similarly analyze the relationship between brightness and power by comparing the device's power consumption across multiple screen brightnesses via the Nexus 4's internal current sensor. The results are presented in Figure 5.2, and suggest a linear trend between power and display brightness. Noise on the signal is attributed to power consumed by other hardware elements. This means that from Figure 5.1 and Figure 5.2, we can treat the OS-indicated brightness levels as linear with measured screen brightness and power consumption.

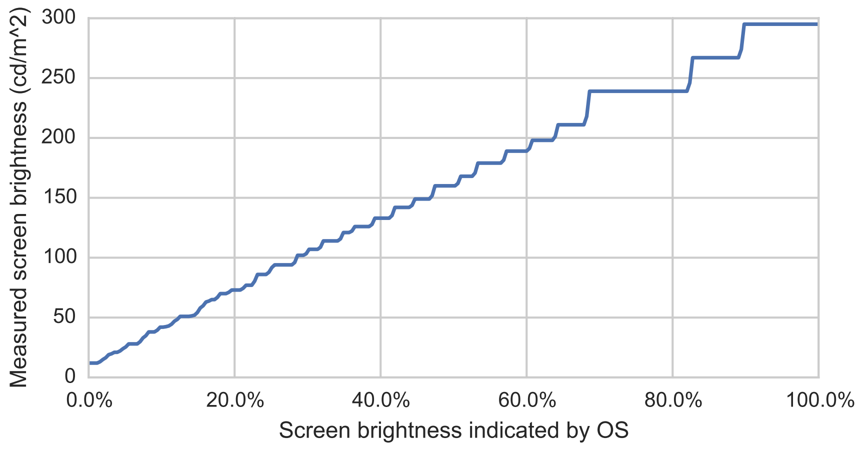


Figure 5.1. Comparison of brightness levels as indicated by the OS to actual brightness levels, as measured by an external light meter. Screen brightness is linear with the OS's indicated levels, although the granularity of brightness drops at higher screen brightness levels.

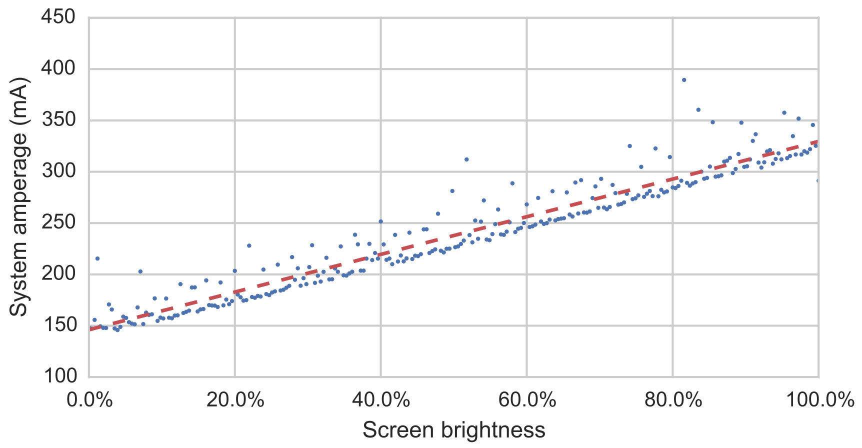


Figure 5.2. Comparison of Nexus 4 current draw at various screen brightnesses with idle system. Blue dots represent individual amperage readings, and the dotted red line represents the linear best fit.

Finally, we wish to note what adaptive brightness system the device uses by default. In the case of the Nexus 4, the brightness levels are adjusted via a static model which translates the ambient light levels to a corresponding screen brightness. This data was retrieved from the operating system, and the continuous model of this data is presented in

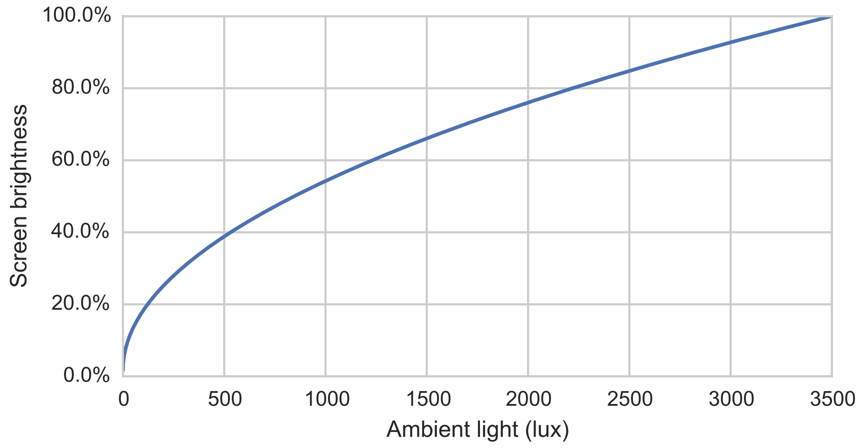


Figure 5.3. Brightness curve of adaptive brightness model on the Nexus 4 device. Screen brightness is linear with the square root of ambient light.

Figure 5.3. According to this model, the screen brightness is linear with the square root of the ambient light levels, reflecting Steven's Law [119].

5.3. Experimental Setup

The goal of this work is to develop a better understanding of how a wide variety of users interact with and respond to mobile displays, and to explore potential power and user experience optimizations in light of this new data. The first two experiments we perform are conducted in a lab environment which allows us to directly control ambient light and screen brightness levels, giving us further insight into the impact that screen brightness and ambient light has on users. We also conduct a third experiment which allows us to analyze how users interact with their devices in typical day-to-day use. The details of these experiments are described below.

5.3.1. Readability Metrics

We introduced the notion of readability in Section 2.1, but thus far have only used the term in the generic sense. For the remainder of this work, we use Relative Visual Performance (RVP) as our target readability metric. RVP is an objective metric which directly measures how quickly and accurately a visual discernment task can be completed. A visual discernment task involves asking users to identify textual glyphs of a specified size; the Snellen chart, which contains letters of descending size and is commonly used by optometrists, is one example of this. RVP has been shown to have a strong dependency on contrast ratio and luminance, as described by CIE [25].

To make an RVP measurement, a user is asked to visually identify a series of characters with a *critical detail* of a known size. A critical detail is a feature in a textual glyph which is required to be properly resolved to correctly identify the glyph (see Figure 5.4) [67]. One test, commonly used with young or otherwise illiterate test subjects, consists of a series of E characters which are tumbled in one of four orientations, as in Figure 5.5. The subjects are asked to identify the orientation of each of the letters in this test. The accuracy and speed at which users can identify these is measured, and RVP is calculated via Equation (5.1). Because RVP is an experimentally-derived metric, it will vary between iterations and users to some degree.

$$speed = \frac{correctly\ identified\ E's}{time}$$

$$accuracy = \frac{correctly\ identified\ E's}{total\ E's}$$

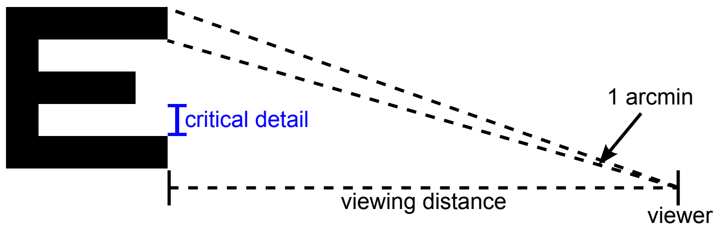


Figure 5.4. Diagram of critical detail size of the letter ‘E’. Critical detail is 1/5th of overall height of letter. 20/20 vision requires ability to resolve 1 arcmin of detail.

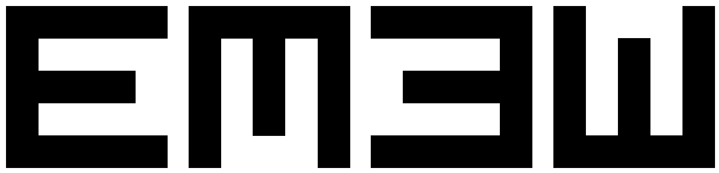


Figure 5.5. A series of tumbling E glyphs.

$$RVP = accuracy \cdot speed \tag{5.1}$$

5.3.2. Screen Readability Study

We wish to understand how RVP varies with screen brightness and ambient light levels across a variety of users by directly comparing RVP to subjective user ratings in identical ambient lighting and display environments. To accomplish this, we analyze readability in this study, and then perform the subjective rating study described in Section 5.3.3. This study was set up as follows.

Users are asked to sit down in front of a Google Nexus 4 smartphone device which is fastened in front of the user. A high-brightness halogen lamp stand with a variable brightness slider is used to control the ambient light levels to simulate the full range of brightnesses that a user may encounter in day-to-day life. This lamp allows us to create ambient lighting environments ranging from a dark room (0 lux) to a sunlit day ($\geq$ 4999 lux). The lamp is placed behind the user's shoulder and aimed toward the fastened Nexus 4, causing reflections which reduce the effective contrast ratio.

As shown in Figure 5.6, users begin with a blank screen which contains a *start* button. Upon pressing that button, users are presented with a grid of 20 E's which are tumbled in either an up, down, left, or right orientation (as in Figure 5.5). Users are asked to touch the E's which are in either an *up* or *down* orientation, while ignoring those in a *left* or *right* orientation. Upon completing this visual discernment task, the user presses the *stop* button, and our system logs the screen brightness, ambient light level, and the speed and accuracy at which the user completed the task. This same task is repeated 60 times (5 discrete ambient light levels, and 12 screen brightness levels for each), with a 15-second gap in-between tasks, each at a different screen brightness and ambient light combination. The ordering of these ambient light and screen brightness combinations is randomized to reduce order-effect bias.

1 arcmin is a generally accepted threshold of detail that a healthy human eye can typically resolve, and is the visual discernment resolution required for 20/20 vision [67]. For this reason, we use 1 arcmin as our target critical detail size, as shown in Figure 5.4. We set this gap size to exactly 2 pixels on our study, which allows us to avoid dealing with subpixel display issues like anti-aliasing. Since the Nexus 4 has a 320ppi screen, we situate users 21.6 inches from the display to achieve the 1 arcmin critical detail gap. Users are given the option of using a larger E if they couldn't discern 1 arcmin of detail, but no users required this.

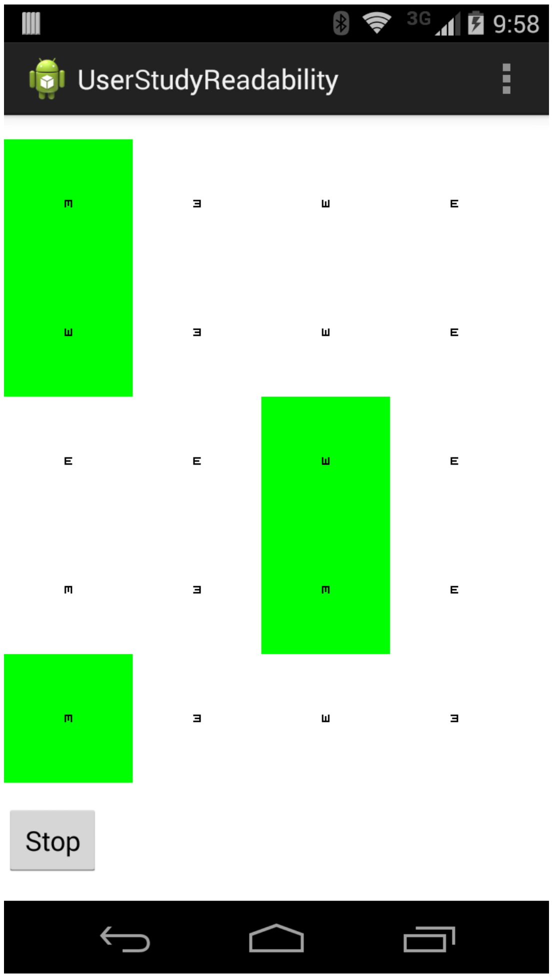


Figure 5.6. Interface used to conduct readability user study. Upon pressing start, a series of E's in random orientations appear. Users are asked to touch the E's which are in an up or down orientation and disregard those in a left or right orientation. Readability is determined from the speed and accuracy at which they can accomplish this.

We ran this experiment on a set of 30 undergraduate and graduate students.

5.3.3. Subjective Brightness Satisfaction Study

In addition to readability analysis, we are also interested in relating user satisfaction to ambient and brightness levels. We use the same lighting setup and the same 30 users as in

Section 5.3.2. We again fasten the phone at a constant distance of 21.6 inches from the users.

This time, however, instead of having the users perform a synthetic visual discernment task, users are asked to read and interact with the BBC News Android application [11]. While they are browsing the articles on the application, a dialog box pops up every 30 seconds, as shown in Figure 5.7, and users are asked to rate how satisfied they are with the current ambient light and screen brightness levels. The options are *much too dim*, *slightly too dim*, *perfect*, *slightly too bright*, and *much too bright.* Users are instructed to interpret the modifier *much* as a lighting situation where the user would likely go out of their way to adjust the screen's brightness. *Slightly* is defined as a situation where users likely wouldn't adjust the device, but it isn't set exactly where they'd prefer. *Perfect* is just as it sounds; users are completely satisfied in this situation.

As in the previous study, after each of these subjective ratings, the ambient light and screen brightness levels are randomly adjusted to one of the 60 discrete ambient light/screen brightness combinations. These ambient light and screen brightness combinations are identical to the previous experiment to allow for direct comparisons between the two studies. This is repeated 60 times, and the ambient and screen brightness levels and the subjective ratings are recorded after each user indication.

5.3.4. Device Usage Study

As a final data collection component to this study, we also want to understand how real users interact with their devices over the course of a day. Specifically, the data we wish to collect includes the average amount of time per day that users spend on their phones,

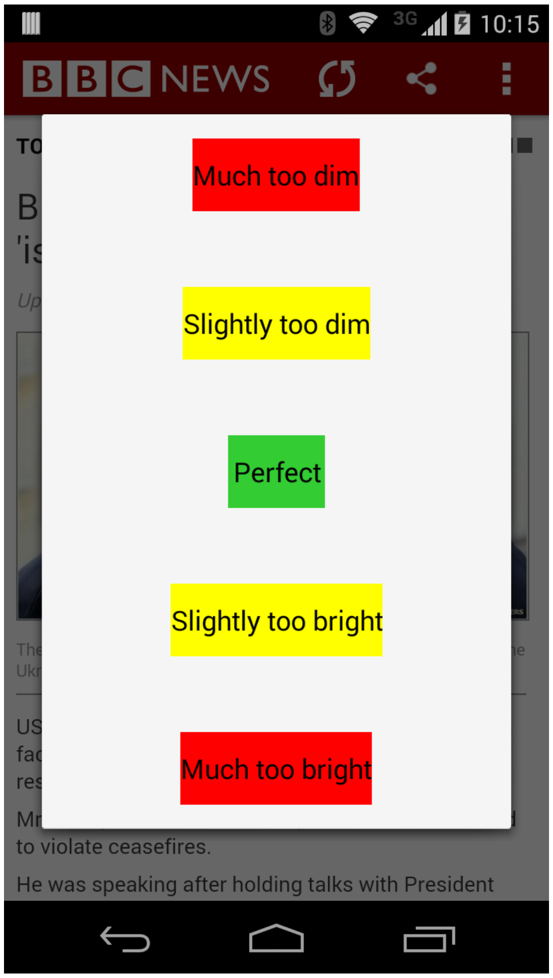


Figure 5.7. Interface used to conduct subjective ratings user study. Users are regularly prompted to provide their satisfaction level with the current ambient light and screen brightness combination while reading the BBC News application.

as well as the ambient light levels that they encounter throughout the day. With this data, we can then integrate any power savings techniques we come up with over time to determine the daily battery impact that these techniques have.

We collected data from 5 individual users over the course of 14 days. We log the times that the users turn on/off their smartphone screens, as well as the ambient light levels that they are in during this time. We poll this data at a rate of once every 5 seconds. This collection system was devised as an Android application which was distributed via the Google Play Store [47] to users from a wide range of geographies, which helps account for bias due to latitudinal location.

5.4. Results

In this section, we describe the analyses of the experiments described in Section 5.3. We analyze the correlation between screen readability and subjective ratings at various ambient light and screen brightness levels in Section 5.4.1. We investigate how well-suited existing default adaptive brightness system are for our sampling of users, and compare this to a calibrated model in Section 5.4.2. We then analyze the potential power savings that we can achieve by degrading the readability and ratings metrics by known quantities in Section 5.4.3. In Section 5.4.4, we present our analysis of the ambient light and device usage study. We then present our readability-aware brightness throttling model in Section 5.4.5. Finally, we give a comparison of the impact that power saving decisions made by brightness throttling systems have on readability in Section 5.4.6.

5.4.1. Screen Readability and Subjective Ratings

In Sections 5.3.2 and 5.3.3 we described the studies that were conducted on 30 users to gather data on users' readability and subjective satisfaction in these lighting scenarios. Now, we average the 30 users' data for each of the 60 ambient lighting and screen brightness combinations. We aggregate the user data study rather than deal with individuals because we want to understand how the average behaves, although individual tendencies are important in final user-facing applications [108, 116, 117].

The aggregated data is presented in Figure 5.8. The readability and ratings data are plotted on separate axis, but the axis are scaled to make them directly comparable. The readability data is presented as raw data; it is a continuous metric, and allows for presentation in this manner. The rating data, however, is nominal; to make it possible to graphically present this data, we convert the nominal ratings to numeric data before averaging each user's data. Any rating of *perfect* is given a value of 3, any rating of *slightly too dim* or *slightly too bright* is given a value of 2, and any rating of *much too dim* or *much too bright* is given a value of 1. This allows us to not only see the relative satisfaction of the users with the screen's brightness, but also see the point at which the ratings peak; either side of this peak would be relatively too bright or too dim.

Figure 5.8 shows that the default adaptive brightness system doesn't align well with either RVP or the ratings peaks in these graphs. This suggests that the adaptive brightness system is not calibrated well on this device. We explore this further in Section 5.4.2. RVP doesn't seem to decline after a certain point like the subjective ratings do, suggesting that although users can read the detail on the device well, it is still not set to an optimal level. This is important to consider as we begin to draw similarities between RVP and the subjective user ratings. Finally, the display's brightness seems to have diminishing returns as it is increased; after a certain point, the screen has to be significantly brightened to achieve just a small improvement in ratings or RVP. This is a potential point for optimization, which we investigate further.

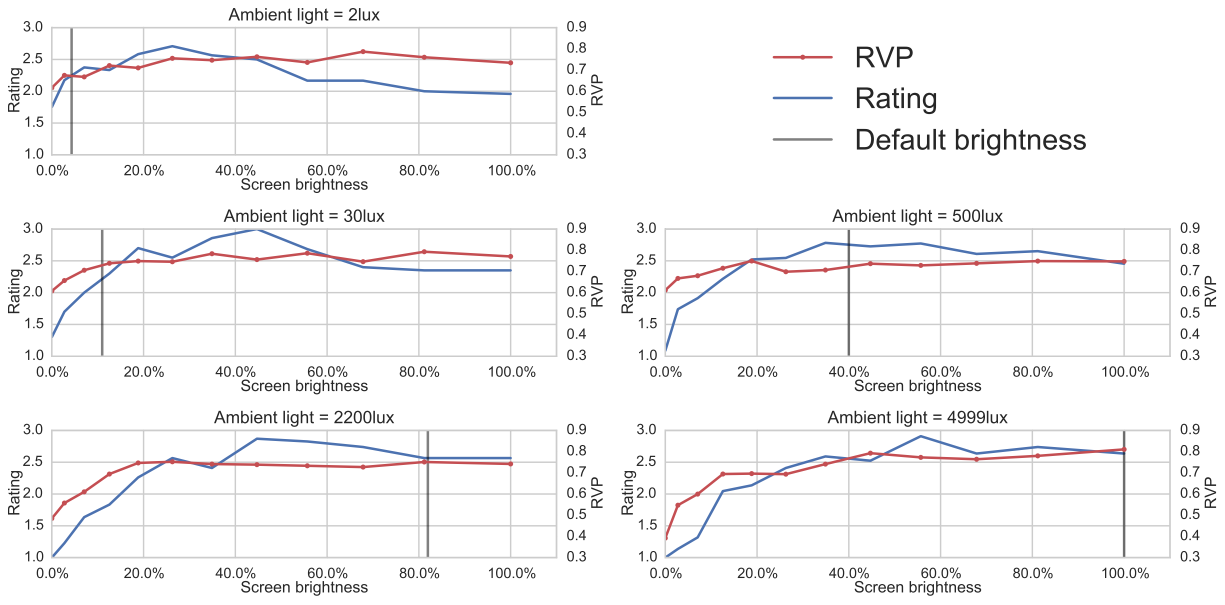


Figure 5.8. Comparison of RVP and subjective ratings at various ambient light and screen brightness levels.

In addition to the overlaid RVP and ratings plots, we analyze the correlation between RVP and the subjective ratings for each ambient level. To perform this analysis, we consider each ambient lighting scenario separately. We then calculate each user's individual correlation coefficient by comparing their RVP and subjective ratings at each screen brightness level and present the aggregated correlation coefficients in Figure 5.9. Readability tends not to worsen when the display is too bright, although the subjective ratings do degrade; because of this, we also create Figure 5.10, which only calculates the correlation between the lowest brightness level and the subjective ratings peak for each ambient level. As these graphs show, there is a strong correlation between the two metrics. However, the correlation is more significant at higher brightness levels. Additionally, the correlation between readability and RVP is even higher when you only consider this up to the maximal ratings point for each ambient level. Because RVP is so closely mapped to contrast ratio

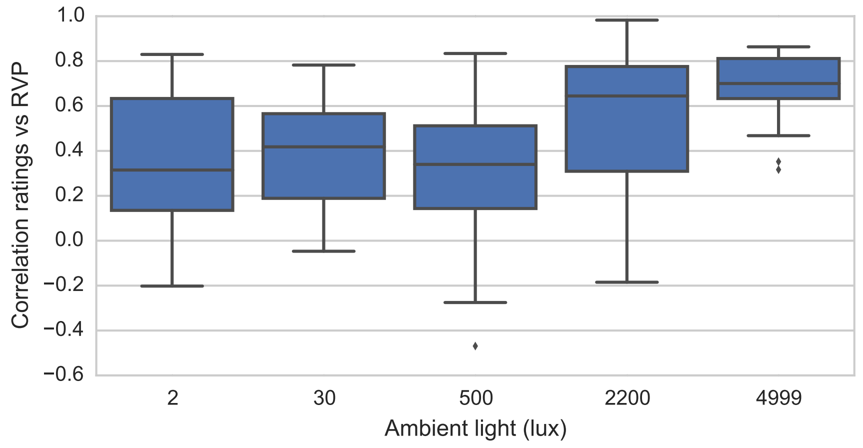


Figure 5.9. Correlation between RVP and subjective ratings at various ambient light levels. Correlation is computed using Pearson's correlation factor for RVP and subjective ratings at selected screen brightness levels across 30 users. Correlation is more significant in higher ambient environments

and luminance [70], this data suggests that it is possible to use objective task performance metrics (such as RVP, in the case of display brightness) as an indication of a system's effectiveness, rather than needing to rely on subjective metrics, which require direct user interaction.

5.4.2. Default vs. Perfectly Calibrated Power Consumption

As shown in Figure 5.8, there is a particular brightness level where increasing the brightness of the screen any further stops causing increased user satisfaction, and actually begins to degrade the user's subjective satisfaction with the system. This phenomenon occurs because after some preferred brightness, the screen is excessively bright for the user and actually begins to degrade their experience [94]. Furthermore, it shows that these ratings "peaks" don't actually always align with the default brightness model's brightness at that ambient light level. This disconnect between the ratings peaks and the adaptive brightness

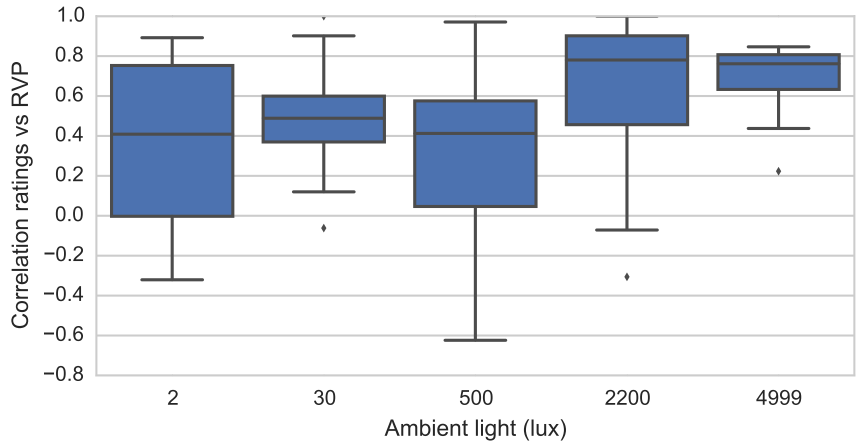


Figure 5.10. Correlation between RVP and subjective ratings. Instead of correlating the full range of brightness levels, each boxplot only contains data between brightness level 0 and each ambient level's ratings peak. This second analysis demonstrates stronger correlation because ratings drop when the screen is deemed to be too bright; overly bright screens do not reduce readability in acclimated eyes, however.

system suggests that the adaptive brightness system isn't well-tuned. This also suggests that if we were to use this "perfect" brightness model instead of the default adaptive brightness model, the display would consume different amounts of power as a result.

To explore this further, we first begin by finding the highest-rated screen brightness level for each of the ambient levels that we explore in our study. Additionally, we find the screen brightness that the default adaptive brightness system would use for each of these ambient lighting environments. Finally, we calculate the difference in screen brightness between the two and apply the power model from Figure 5.2, which gives us how much more or less power is consumed by the display at the various ambient light levels. This data is presented in Figure 5.11.

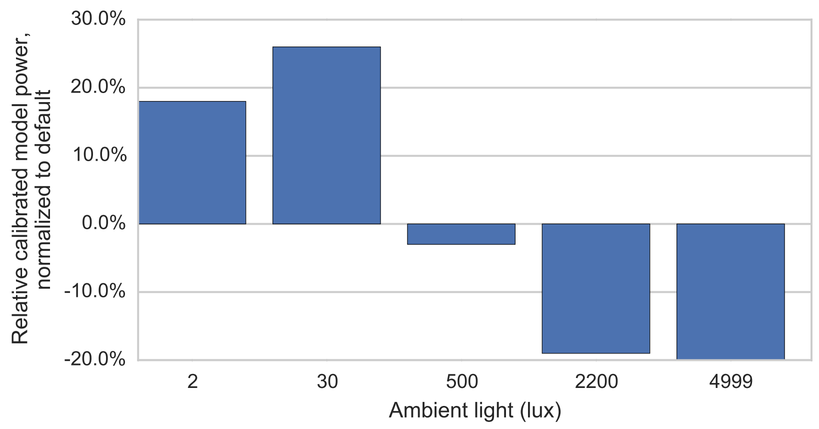


Figure 5.11. Relative power consumption of a perfectly calibrated adaptive screen brightness model (screen brightness at the highest average rating), normalized to the default adaptive brightness system's power consumption.

As the figure shows, users actually preferred the display's brightness to be set higher than the default model's brightness at lower ambient light levels. Additionally, users preferred the display to be dimmer than the default model at higher ambient levels.

5.4.3. Power Savings and Readability Degradation

In Section 5.2, we presented a series of graphs which relate OS-specified screen brightness values, measured screen brightness, and screen power consumption. These three sets of data together allow us to determine the impact on system power consumption that any change in screen brightness has. Furthermore, Section 5.4.1 shows us exactly how much of a change in screen brightness impacts users' RVP at a given ambient level. These individual pieces of data can be used in tandem to gauge the impact that improving or degrading screen readability has on screen power consumption.

From Figure 5.8, we analyze the change in screen brightness that results from degrading the 30 aggregated users' readability at a given ambient level by a known percentage. The results of this analysis are presented in Figure 5.12. As this figure shows, a significant amount of power can be saved at the higher ambient light levels with a relatively small readability or ratings degradation. Also, the amount of power that can be saved at the lower ambient light levels is much smaller. This trend is reflected in the work done by Stevens [119], which suggests that human perception generally does not follow a linear trend. For instance, the human eye can detect a tiny flicker of light in a dark room, but that same flicker would be imperceptible in a brighter environment. We use this result in our further analysis, which allows us to consider the human visual system in any power savings decisions we make. There is slightly less potential power savings for a given readability level at 4999 lux than there is at 2200 lux. This is likely due to the initial default adaptive brightness levels; this suggests that the 4999 lux screen brightness levels better matched user requirements than at 2200 lux, and is reflected in the slightly lower power savings.

5.4.4. Ambient Light Residency Analysis

In Section 5.3.4, we outlined a study in which we gather ambient light data from a sampling users to get a better idea of how users interact with their devices. Specifically, we wanted to discover the daily averages of how long users' screens were active and what ambient light levels the users encountered. For this analysis, we combined all of the users' ambient residency durations. We then calculated the CDF of these ambient lighting environments over time, as presented in Figure 5.13. We note that the daily on-screen duration average is 3.87 hours.

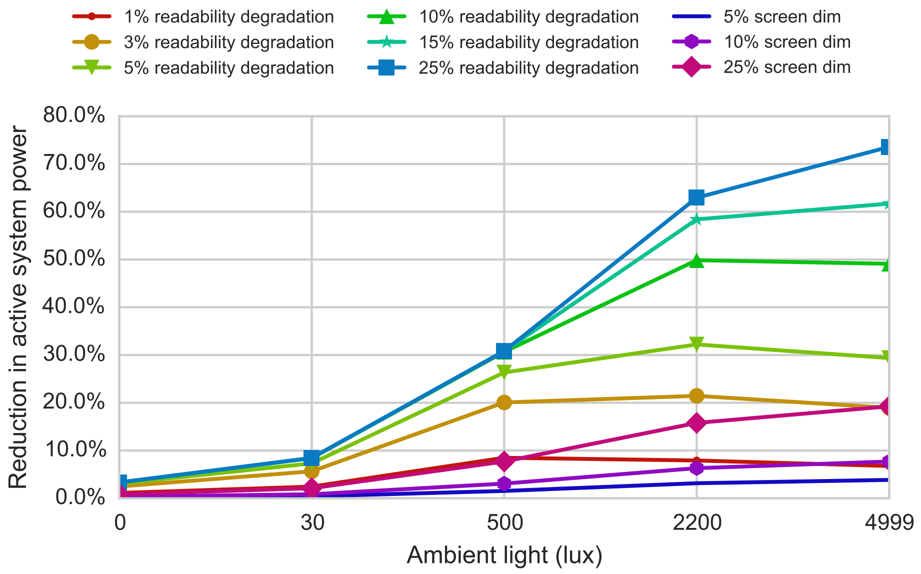


Figure 5.12. Amount of power that can be saved at a specific amount of readability degradation, across selected ambient light levels.

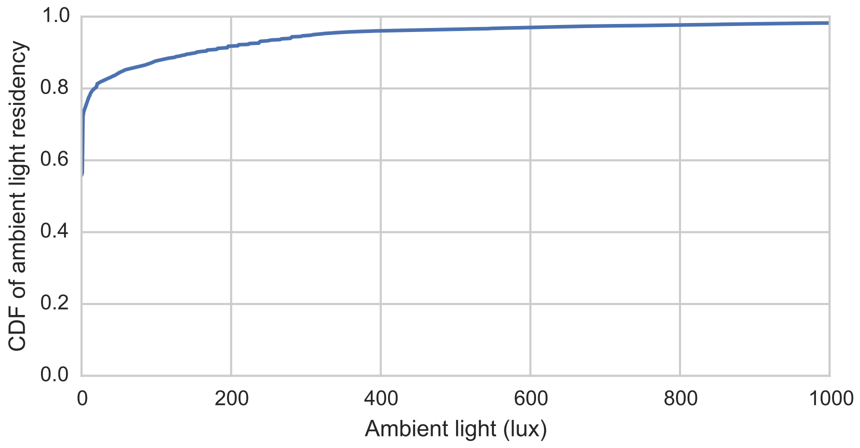


Figure 5.13. Amount of time that users spend in various ambient light levels with their screens active.

Dim environments ($\leq$ 30 lux) are shown to make up about 80% of the total on-screen time. Typically, only interior and nighttime environments produce this range of ambient light levels, and so this suggests that users spend most of their on-phone time indoors. Because most of the time is spent at lower ambient light levels, if you were to use the perfectly tuned adaptive brightness model as discussed in Section 5.4.2, the system would consume more power since this model consumes more power than the default at lower ambient light levels (and vice-versa). However, this would improve the overall satisfaction levels, effectively making this a design decision.

5.4.5. Readability-Aware Brightness Model

Thus far, we have presented results which characterize the relationship between user satisfaction, readability, and power consumption. In the remainder of this work, we focus on the actual design and characteristics of a dimming algorithm which is better tuned to reflect actual user readability requirements. Our end goal is to be able to maximize the time-weighted readability average over a period of time. We learned in Section 5.4.3 that the brightest ambient regions provide the most significant power savings per readability degradation level. Hence, in order to maximize the time-weighted readability, we will throttle the screen's brightness more significantly in higher ambient-lit environments.

We begin with our calibrated readability curve, which is a static curve of screen brightness vs ambient light. For each amount of desired power saved (we generate data between 1% and 8%), we then generate a readability-aware brightness curve. For each of the discrete ambient levels that we studied, we iteratively dim the brightest ambient regions (while enforcing monotonicity of the screen brightness) until our time-averaged power savings are met. This shape is most closely matched to a root function (and is supported by the evidence from Stevens [119]), so we generate the line of best fit through these discrete points. We present these generated curves in Figure 5.14.

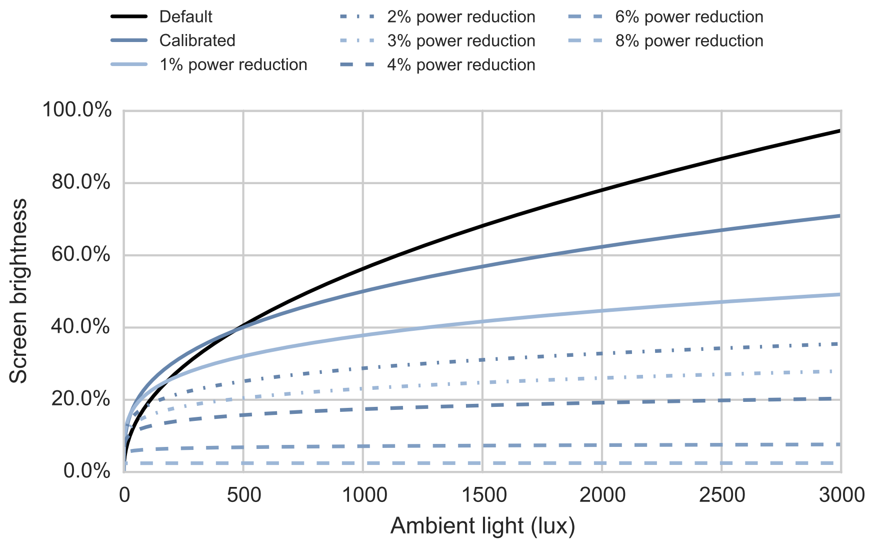


Figure 5.14. Comparison of adaptive brightness curves. The depicted curves include the default adaptive brightness algorithm, as well as the readability-optimized curves for selected power savings levels.

These resulting curves have a number of interesting characteristics. First of all, the dimmest ambient levels tend to actually be slightly brighter than the default algorithm, suggestion that the default brightness algorithm isn't properly calibrated. The highest power reduction curves are nearly flat; these approach the maximum power that can be saved on the screen, and so the brightness is aggressively low at all ambient levels. The less significant power throttling curves look more like a traditional root function.

These graphs represent how the proposed brightness algorithm will behave in terms of screen brightness and power savings; in the following section, we examine how they impact the time-weighted readability levels.

5.4.6. Power Savings and Readability Comparison

We now analyze how much readability is reduced for a given amount of power savings for a selection of brightness dimming schemes. These dimming schemes all reduce the brightness of the display in order to achieve a power reduction target. How these schemes differ, however, is how much they dim the display when they are in various ambient lighting regions. Because we are evaluating these schemes in a time-weighted fashion, a scheme which dims the display more aggressively in dim environments will have to dim the display less in brighter environments to compensate for this, and vice-versa. These schemes have been observed in various subsystems which throttle display power consumption.

- **Constant:** throttles the screen's brightness by a specified amount, regardless of the current ambient light or display brightness levels. The resulting effect is that the power is reduced by the same amount at any ambient light level. For example, this scheme might reduce a display at 100% brightness to 90% brightness, and a display at 20% brightness by the same absolute amount, to 10%. This scheme is effective when there is a constant required amount of power savings.
- **Fractional:** instead of reducing the brightness by a constant amount, the brightness is reduced depending on the screen's starting brightness. If this scheme reduces a display at 100% brightness to 90%, it would reduce a display at 20% by the same relative percentage, to 18%. This system will reduce the power more in brighter environments, typically leading to higher power savings.
- **Optimal:** considers the user's readability curve, and first dims the regions which give the best power vs readability trade-off, as introduced in Section 5.4.5. This

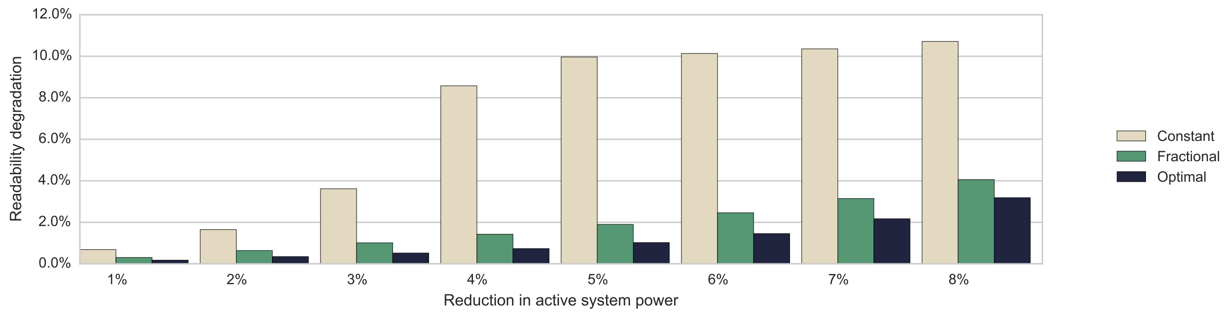


Figure 5.15. Comparison of the impact on readability that various dimming schemes have at selected power savings thresholds.

scheme maximizes the time-averaged readability at a given amount of power savings.

To perform this analysis, we determine how much the average readability is degraded at various power savings levels. We use the data from Figure 5.13 to apply the ambient light residencies data which gives us a time-weighted average impact on readability.

As Figure 5.15 shows, these schemes have significantly different time-weighted average readabilities. The constant scheme has the worse time-weighted readability average. This makes intuitive sense; since the lower ambient level regions are much more sensitive to brightness changes, a large brightness decrease in these has a huge negative impact on readability. The fractional and optimal models have better power reduction vs readability reduction performance, as they dim the displays more severely in the brighter ambient lighting environments, and vice-versa. At an 8% power reduction target, the time-weighted average readability drops by 10.7% for the constant model, 4.1% for the fractional model, and 3.1% for the optimal model, a 21.5% improvement over the fractional model. This result shows that an optimized power model is able to deliver a significantly better average user experience at a given power savings level.

Being able to directly understand the impact that a display brightness degradation decision has on the user is useful. Instead of arbitrarily making system-level decisions which directly affect the end user, these curves allow the system to first consider the user impact when throttling the device's power. Secondly, it allows for the creation of models (like our optimal model), which are optimized to maximize the user's satisfaction with the system for a given set of constraints.

It is important to note, these methods are only applicable when saving power over longer periods of time. If the system decides that it needs to reduce the amount of power being consumed immediately (e.g., a high-priority thermal emergency), it may have to do this without being able to consider the impact on readability. However, even in this situation, the system is aware exactly how much of an impact such a decision has on the user, which enables better-informed power saving decisions.

5.5. Conclusion

In this work, we ran a series of user studies to analyze subjective user satisfaction and objective readability metrics on mobile LCD displays in a variety of ambient lighting environments. From these studies, we were able to find a strong correlation between our subjective satisfaction and objective readability metrics; this is important when attempting to design a system which is aware of the user's satisfaction with a system, but doesn't interrupt the user for subjective ratings. We also found that the adaptive brightness systems currently in use are not necessarily well-tuned. Furthermore, from this data, we found that many brightness throttling systems do not consider the impact on the user when they are active. By taking this impact into consideration, we were able to create a display power throttling system which achieves a 21.5% higher average readability and user satisfaction at a given level of power reduction over a period of time compared to a naive solution.

Having an understanding of the impact that a user-facing system decision has on the user is important, as this makes it possible for the system to optimize these power decisions along with the user's satisfaction level. However, the best way to get this information is by directly asking the user. Unfortunately, this is an invasive process which doesn't fit well in the user's work flow. In addition, subjective ratings can be noisy. Because of this, devising objective models which are closely correlated to subjective satisfaction metrics is a promising way of including the user in the system optimization loop, allowing for the system to achieve its own power and performance requirements while maximizing the user's satisfaction in the process.

CHAPTER 6

System-Level Performance Bug Localization Via Causality Analysis

6.1. Introduction

Computing system complexity is growing rapidly due to increases in computation power and wider adoption of these systems. For example, even computing platforms such as smartphones and smartwatches, which are commodity electronic devices, are very complex and consist of a huge number of software and hardware components. The core of these systems are formed by System-on-Chips (SoCs), which are themselves an assembly of multicore processors with a number of hardware accelerators, such as GPUs and MMUs, which are interconnected using on-chip networks and heterogeneous communication protocols. In addition to the SoC, these platforms have other important hardware components, such as the radio, GPS, and a large number of sensors. On such a complex system, performance problems including dropped calls, animation lags, or poor frame rates, are notoriously hard to debug [24, 71]. Since the source of a performance bug might lie in one of many different components of a complex system, or in a particular interaction scenario among both hardware and software components, debugging performance problems is very challenging. The intermittent nature of performance bugs adds an additional dimension of difficulty to this problem.

The importance of finding these performance bugs is escalated by the trend toward a high priority on the user experience in recent years. Modern smart devices are highly interactive, and their workloads are often computationally intensive. Interactive performance problems are readily noticed by users, and this can impact user sentiment, which makes this a concern for device manufacturers. Furthermore, these devices are also often battery and hardware-constrained, which leaves only a small amount of room for error in maintaining satisfactory performance, since significant hardware over-provisioning is generally not an option. This leaves the domain in a particularly tenuous position: with difficult-to-identify, high-impact, intermittently triggered performance problems.

To address these concerns, we propose a novel approach to automatically localize system-level performance bugs. This approach relies on observing system behavior, and does not require prior detailed information about how the system components function and interact with each other. While such prior information can be exploited to aid diagnosis, inclusion of prior information as a debugging bias in our framework is limited to identification of *features* of traces that are likely to be relevant. Examples of features include total runtime of specific OS processes, sequence of wake-up calls of different processes, memory latency observed in hardware, and average instructions per cycle (IPC) of processors. From passively observed traces of the system, we identify *features* of the traces that distinguish traces without performance bugs and those with bugs. This identification is done using causality analysis on the feature space. We use Lewis's counterfactual notion of causality [82] using standard machine learning techniques of decision trees [74] and support vector machines (SVM) [30].

In this work, we specifically focus on the challenge of debugging display and graphics performance of Android platforms to demonstrate the effectiveness of our approach. This choice is guided by two reasons.

Firstly, platforms such as smartphones are user-facing computing devices, and graphics frame rate is one of the defining performance metrics and directly impacts the user experience. These systems support advanced graphics to enable games and high resolution videos, and thus require significant computing power. However, the battery constraints and form factor limit the computation power and hardware support on these systems. Thus, a delicate balance needs to be maintained between the graphical performance on one side, and hardware cost and power consumption on the other. Display performance depends on this balance, and a subtle bug in interaction between different software and hardware components can cause the performance to deteriorate [66].

Secondly, the graphics and display subsystem is an excellent example of a complex system that involves interaction between the application software, operating system, firmware, and hardware. An observed performance bug that manifests itself as frame drops might arise due to many different reasons, such as a bug in application code, a problem with OS scheduling algorithms, interaction between firmware power management algorithms and OS scheduling, or some incorrect assumption regarding the hardware accelerator made by the OS scheduling algorithm for a particular firmware power-saving state. While the more obvious sources of bugs (such as an application coding error or a flaw in the hardware accelerators) can be discovered using standard verification and validation techniques, the most challenging bugs are issues arising from interaction between different facets of a system. Another dimension of complexity emerges from the component heterogeneity. For example, the same software stack might be expected to run on a wide set of different hardware, and only some of the hardware might have dedicated accelerators for certain tasks. Consequently, similar performance degradations in different systems might be due to completely different reasons. Thus, the challenge of localizing performance degradations in complex systems is clearly demonstrated in this specific real-world system.

Our approach is implemented through observation hooks at the operating system level, and hence, the implementation can be used on any platform running Android. The observation of the Android system utilizes `atrace`, which is a system-level trace generator included in Android OS distributions. We project this trace to the feature space, and then perform causality analysis to detect the reason behind performance problems. We tested our system on a variety of workloads and we were able to accurately identify causes of performance problems (reduction in frame rate) from a variety of sources including user interaction, application programming bugs, system memory bandwidth contention, and the CPU power management governor.

In summary, we make the following three novel contributions in this chapter:

- We formulate black-box performance debugging as a problem of learning conditions over trace features, and present a solution using decision tree learning techniques [74].
- We provide an alternative SVM-based [30] approach to localize causes of performance problems.
- We demonstrate the effectiveness of our approach for debugging Android's graphics and display subsystem on a variety of workloads.

The chapter is organized as follows. We first present a theoretical foundation of our work and discuss how system debugging, and particularly performance debugging, can be done using causality analysis over system traces in Section 6.2. We also present the decision tree and SVM-based approaches to solve this problem. This is followed by description of a real-world scenario of debugging performance problems in Android's graphics and display system, which is the focus of this chapter. We then describe the implementation of our debugging methodology in Section 6.3, which consists of mapping the observed traces to a feature space under manual guidance, followed by automated causality analysis using SVM and decision tree inference. In Section 6.4, we describe each of the individual workloads that we analyze using our system, as well as the result of performance bug localization on these workloads, and conclude in Section 6.5.

6.2. Performance Debugging and Causality Analysis

In this section, we present a debugging approach based on passively observing system state. An automated monitoring system or human user can be used as an oracle to mark system traces as good or bad, depending on if the trace contained performance problems. This is easy for system-level performance bugs, since performance criterion are often well-defined metrics which can be measured, and violation of some expected level of performance can be marked as a problematic trace. For example, frame rate can be used to measure performance of a graphics and display system. If the frame rate drops below a threshold, say 60 frames per second (FPS), we can mark that frame as one with performance problems. Instructions per cycle (IPC), observed network transmission rate, and delay in registering user inputs are other examples for measuring performance of computing platforms. This allows for a highly flexible definition of a performance bug, allowing not only user-specified metric thresholds, but also agile selection of the metrics themselves. Thus, the first step of our debugging approach is to gather system traces and mark them as good or bad, based on some appropriate performance metric.

A system trace is simply a sequence of events from different parts of the system. For a computing platform, such events could be the start and end of processes, work requests to hardware accelerators, network traffic, interrupts, memory allocations, or user interaction events. Given a trace of these events, we project this to features of interest for the debugging problem. For example, if process duration is a feature of interest, it can be computed from the scheduling events observed from the OS scheduler. The identification of features and projection of traces to the feature space also helps define the space of possible explanations. The features correspond to different components, facets, and interactions in the system. While the selection of features is somewhat manual, domain expertise can be used to come up with these features, as we illustrate for Android's graphics and display system in Section 6.3. The next step is to determine a causal relation between the features and the performance problem.

The traces for system debugging are discrete-time traces of events and features arising out of a deterministic system (albeit, with partial observability) and hence, we select the Lewis's counterfactual notion of causality as the basis of our approach. The basic idea of counterfactual theory of causation [82] is that A causally depends on B, if and only if the following counterfactual conditional holds: *If event B had not occurred, event A would not have occurred, and if event B occurred, then event A would necessarily occur.* Causality, then, corresponds to a chain of causal dependency relations. The theory extends to multi-event causes directly by considering subsets of events instead of a single event. A causally depends on B and C, if and only if absence of either event B or C would mean that event A does not occur; if both event B and C occur, then event A must necessarily occur. There is an inherent asymmetry in this definition of causal dependence. This distinguishes causality from correlation, and also ensures a direction of causality.

In our debugging context, we are given a set of traces $\mathrm{T} = \{\tau_1, \tau_2, \ldots\}$, each of which are labeled as good or bad using some labeling oracle $\texttt{oracle} : \mathrm{T} \rightarrow \{\texttt{good}, \texttt{bad}\}$, which identifies whether the trace corresponds to a good run, or to a run with performance problems. We have features which denote events (Boolean absence/presence), as well as multivalued features such as time stamps of event occurrences, duration of processes, and counts of interrupts. Now, we can map each trace τ_i to a vector of feature values: $f_i^1, f_i^2, \ldots, f_i^n$ where f_i^j denotes either a multi-valued feature or Boolean event occurrence in trace τ_i. The goal of debugging is to discover a set of features $f^{j1}, f^{j2}, \ldots$ and thresholds $c^1, c^2, \ldots$, such that some Boolean combination $\mathcal{B}(f^{j1} \geq c^1, f^{j2} \geq c^2, \ldots)$ is a cause of the poor performance in the sense of counterfactual theory of Lewis:

$$\forall \tau_i \in \mathrm{T} \; \mathcal{B}(f_i^{j1} \geq c^1, f_i^{j2} \geq c^2, \ldots) \textit{ iff } [\texttt{oracle}(\tau_i) = \texttt{bad}]$$

Existing formal synthesis from examples [64, 63] and inductive learning [4, 85] techniques can be used to generate the Boolean combination $\mathcal{B}$ from the labeled featured traces. However, these learning approaches have scalability problems, and they are not robust to errors in observation. These techniques also require active experimentation to collect interesting traces of the system for the oracle to label so that these can be used for learning. Our target is to debug systems with hundreds of features and thousands of passively observed traces. Furthermore, the system traces might have measurement inaccuracies. Thus, we focus on using scalable machine learning techniques which are robust to noisy observations.

6.2.1. Cause Inference By Learning Decision Trees

We can consider the traces as separate data points, where each data point τ_i is a vector of features $f_i^1, f_i^2, \ldots, f_i^n$. Now, identifying the cause of poor performance reduces conceptually to finding a classifier that separates `good` and `bad` traces in this feature space. This problem can be solved using existing techniques for decision trees [74].

We briefly sketch the C4.5 algorithm of Quinlan et al [74] used to solve this decision tree learning problem, due to space constraints. Decision trees are learned based on gain ratio — an information theoretic criteria. A feature f^j which partitions the traces into subsets with most homogeneous labels is chosen such that the subtrees from this feature constraint, $f^j \geq c_j$, have least entropy. The threshold for multi-value features in the feature constraints is selected from the values observed for this feature among the different data points (traces). This is followed by a similar selection of feature constraints to subdivide the subtrees, and this is continued till all traces in a subset are either `good` or `bad`. Thus, iteratively, a decision tree with feature constraints $f^j \geq c^j$ as internal nodes is learned such that all the traces satisfying constraints from the root node to a leaf node are either `good` or `bad`.

The conjunction of conditions on internal nodes from root to a lead node defines the path condition for that leaf node. Now, the Boolean formula for the cause of `bad` traces is the disjunction of all path conditions leading to a leaf node with all `bad` traces, and is

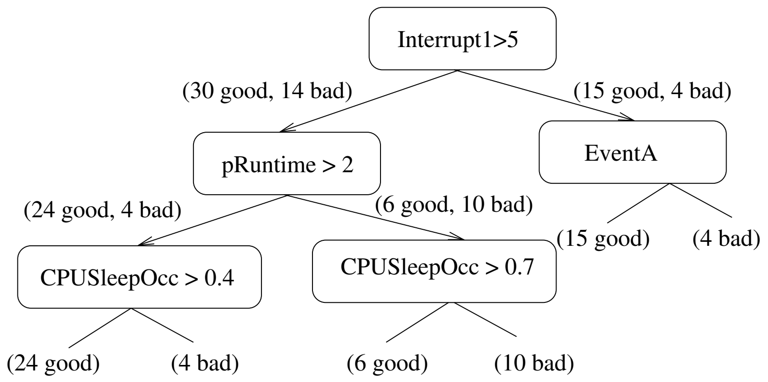


Figure 6.1. Example decision tree: The right branch denotes that the constraint in the node holds, and vice-versa.

the explanation for the performance problem. The features in this formula correspond to potential bug locations. We illustrate this with an example in Figure 6.1. Intuitively, this tree localizes the fault to interrupt1, eventA, pRuntime and the CPU power management system. The original traces could have contained many other features which were not needed to classify the traces into good and bad, and hence, can be ruled out as possible source of performance problem using counterfactual theory [82] of causality.

6.2.2. Cause Inference Using SVM

The features in performance debugging systems often have a large number of multi-valued variables. Some of these are numeric variables, and hence, learning trees requires a search over a large number of possible candidate constraints for each internal node. If we restrict our goal to only identifying the features which participate in causing the performance problem and not learn the actual Boolean structure $\mathcal{B}$, then we can use the more scalable approach to solve this problem using SVMs.

We provide a brief description of SVM [30] before showing how to use it to obtain features which correspond to cause of performance problems. Let us assume that the features have been normalized to have a range $[0, 1]$, and the learned decision function is $\sum\limits_{j \in \mathcal{J}} w^j f^j + w^0$, where $\mathcal{J}$ is the set of features which form the support of the decision function. For all the `bad` traces τ_i, $\sum_{j \in \mathcal{J}} w^j f_i^j + w^0 \geq 1$ and for all the `good` traces τ_i, $\sum_{j \in \mathcal{J}} w^j f_i^j + w^0 \leq -1$ Clearly, the features not in $\mathcal{J}$ can not be responsible for the performance bug. Further, let $\mathcal{P} \subseteq \mathcal{J}$ such that the coefficients w^j for all $j \in \mathcal{P}$ are positive, and $\mathcal{N} \subseteq \mathcal{J}$ such that the coefficients w^j for all $j \in \mathcal{N}$ are negative.

We can now interpret the cause of a performance bug as follows: if the feature is a multivalued feature, the features in $\mathcal{P}$ are the features for which increasing the value causes the performance problem, and if the feature is a multivalued feature, the features in $\mathcal{N}$ are the features for which decreasing the value causes the performance problem. For some features, such as duration of processes, we need to only consider the impact of increasing duration, i.e., when the coefficients are positive. Furthermore, since the range of features is normalized, the weights w^j can be used to order the features in terms of their significance. The causality analysis using SVM assumes that the features causing performance problems are monotonic. For example, if a performance problem occurs when the runtime of a process is above a threshold α, then the performance problem would not go away if the process runtime exceeded some higher threshold $\beta > \alpha$. Such a monotonicity assumption is not restrictive, as many systems, including our application, satisfy it. A simple example of SVM-based debugging is presented in Figure 6.2, where increase in countA or pRuntime causes the performance problem.

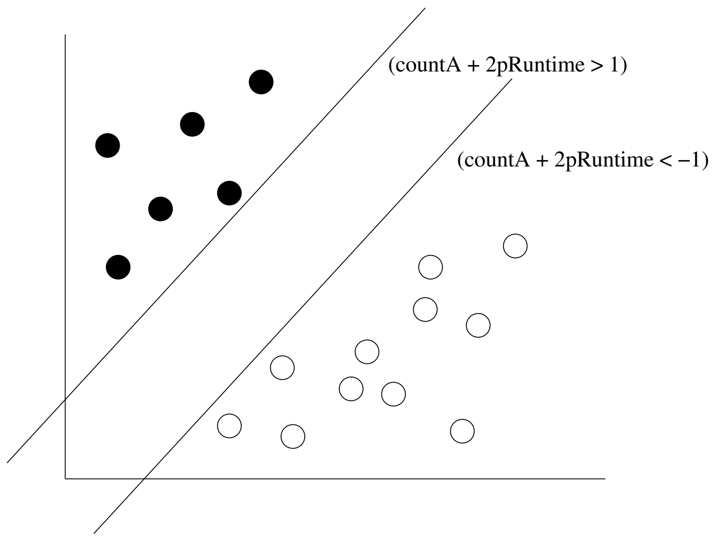


Figure 6.2. Example SVM: Filled circles denote the `bad` traces and unfilled circles represent `good` traces.

Thus, the goal of finding the cause of a performance problem reduces to learning a classifier (decision tree or SVM) on the set of `good` and `bad` traces, which are traces with and without performance problems, respectively. An astute reader might notice that decision tree classifiers would be very effective when most of the features are Boolean, while SVM would be more effective in cases with features which are multi-valued, such as counts and durations. The classification techniques identify features which separate the `good` traces from `bad` traces, and these features correspond to localization of performance problems. Thus, we effectively use classification techniques here as feature selection algorithms, such that the selected features are part of the cause for performance problems. This reduction of debugging to solving standard classification problems enables us to develop a scalable and effective performance debugging technique to analyze systems using only passively observed traces.

6.3. System Description

In this section, we describe the implemented system that we use to localize sources of performance bugs in our system. We first describe the method that we use to collect traces, and then we explain how to convert our raw trace data into individual frames and descriptive features followed by the causality analysis techniques using decision trees and SVM.

6.3.1. Android System Tracing

Our implementation uses `atrace`, which is a system-level trace recording utility provided by Android. `atrace` unifies instrumented kernel and platform-level information into a single trace, and thus, allows us to gain significant insight into the system behavior. `atrace` builds on the Linux-based tracing system `ftrace`, which was originally intended as a kernel function tracer, allowing users to trace entry and exit from functions inside the Linux kernel. There are hundreds of events that are available by default in the kernel, and we use a subset of these events in our system. The second major subcomponent of `atrace` is the Android-specific OS tracer. This component is comprised of a number of instrumented functions and statements within the non-kernel Android subsystems. Android is heavily instrumented by default, and standard macros allow developers to dump additional debug information to `atrace`. This gives us significant observability into the non-kernel portion of the Android OS over time. Since Android is open-source, this also allows adding additional observation hooks. The CPU overhead of the tracing system is typically very lightweight, with CPU utilization of under 2% on a single core.

6.3.2. Features and Instances

In this section, we describe the raw data that we use from `atrace`, as well as how we extract features from this raw data. The scheduler, which denotes when threads are active and inactive on the CPU, is instrumented in the kernel for `ftrace`. This allows us to identify exactly when any given thread is active on the CPU, and the duration of that thread being active. Additionally, the Linux kernel is instrumented to provide information regarding when it begins and ends handling hard and soft interrupts. From this information, we are able to determine what the CPU is processing at any given point in time. `atrace` also can provide information regarding device frequency states via `ftrace`. Specifically, the CPU and GPU power states are included in these traces. The kernel's memory management system is also instrumented, which allows us to track the allocation and deallocation of memory pages in the kernel and userspace.

Since the Android platform is also heavily instrumented for debugging with `atrace`, we can gather significant information regarding various subsystems in Android. Of particular interest are the instrumented Surfaceflinger functions; these allow us to see exactly what phase of execution the graphics subsystem is in. Furthermore, this allows us to see the state of the various BufferQueues in Android over time. In addition to this, Android's input subsystem is instrumented, which lets us analyze the time and frequency of user interaction with the system. Table 6.1 presents different classes of features that we extract from `atrace` data. The number of features will vary by workload, but typically range from roughly 70 to 220.

To organize this data into individual traces, the rendering of each frame is considered to be a single trace, and so we group the features from that timespan accordingly. In

Table 6.1. Numeric features extracted from `atrace` data. These features are included as inputs to analysis system. Each instance contains data during a single frame.

Feature	Description
Process duration	For each unique process, amount of time spent in run queue
Process existence	For each unique process, Boolean (0/1) reflecting if the process was active
IRQ count	For each device or soft IRQ source, the number of times the IRQ was triggered
IRQ duration	For each device of soft IRQ source, the amount of time spent handling the IRQ
Average device frequency	Average DVFS frequency for each device with scalable clock frequency available
Power state	Power management state of the cores
OS function count	For each instrumented OS function, the number of times the function was called
OS function duration	For each instrumented OS function, the amount of time spent in that function
RAM activity	Number of pages allocated and deallocated
User interaction count	The number of individual user interactions (touch, swipe, or button presses)
Frame rendering time	The amount of time spent rendering the frame

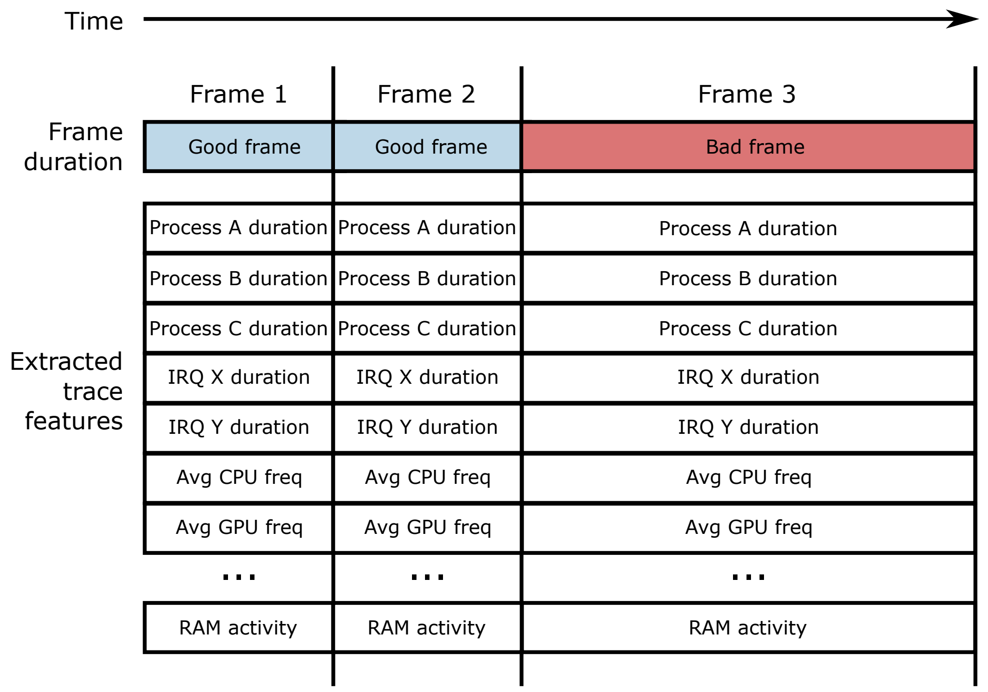


Figure 6.3. Procedure for splitting trace data into individual frames and assigning feature labels. Each instance contains system state during a single frame, as well as a binary indicator of frame duration. Frames that are below the source's nominal frame period are labeled "false" and vice-versa.

addition to these features, we label each frame in a binary fashion — as `good` if it met the timing requirements with respect to the refresh rate (60/30/24 Hz), or `bad` if it is dropped. Figure 6.3 depicts how the extracted instance and features are organized.

6.3.3. Data Analysis and Bug Localization

We now describe the system we use to localize performance bug sources using causality analysis, as presented in Section 6.2. Our implementation for decision tree directly uses the J48 implementation in WEKA of C4.5 algorithm [74]. We use a feature subset selection wrapper [73] around the J48 implementation to recursively remove features which perform poorly, until an optimal feature subset is obtained. Our implementation for SVM technique is built around scikit-learn [99], a python machine learning library. Feature selection in SVM is done using the SVM-Recursive Feature Elimination algorithm (RFE) [52]. To analyze a given trace, we take our frame instances, features, and classification labels as described in Section 6.3.2, and add them as training data to the SVM model. The resulting model coefficients are then sorted by their magnitude. We then begin iteratively removing the least significant of the coefficients and measuring the model's prediction accuracy after each feature is removed. We continue removing features until the model's prediction accuracy begins to drop. At that point, we know that the remaining features are relevant to making accurate frame performance predictions. In Section 6.4, we present evidence for the effectiveness of system-level bug localization findings for a variety of different workloads. We present an analysis using both SVM and decision trees for each workload we consider.

6.4. Results

In this section, we present evidence for our system's ability to localize the source of various classes of performance bugs. We include the features selected by our SVM and decision tree implementations, and also present tables describing the SVM's individual feature coefficients. These workloads contain sources of commonly encountered bugs, and we validate our system's findings in each of the following subsections.

6.4.1. Android Platform-level Bug

One common source of performance bugs in embedded devices is from bugs in the operating system itself. These bugs are particularly difficult to identify because they lie outside of the programmer's typical development work flow. Programmers and users often have partial visibility to the OS, in the case of a closed-source operating system; even in open source operating systems, standard debuggers and profilers operate at the application level rather than at the system level. Programmers can manually connect to system processes with a debugger, but there are enough system threads to make this hugely time consuming, and correlating this data with the application code is a non-trivial process. Hence, a performance bug in the operating system can be particularly difficult to identify.

To validate our system's ability to successfully identify a platform-level bug in Android, we introduce a performance bug into Android's Surfaceflinger application. This bug occurs in Surfaceflinger's queueBuffer function and causes the function to intermittently run longer than normal, which can result in dropped frames due to blocking Surfaceflinger's page flip logic. To generate data on this bug, we play a video with a constant 30Hz frame rate. Due to the performance bug, we expect some of these frames to take longer than the standard 33.3ms period to display. We gather a 20-second long trace via `atrace`, and then analyze that data with our analysis and feature selection program.

The results from our bug localization system are presented in Table 6.2. Remember, these results contain the highest-magnitude coefficients; positive coefficients are positively correlated with `bad` (long-running) frames, and vice-versa. The SVM analysis was able to successfully identify the amount of time spent in Surfaceflinger's queue function as the primary contributing feature to long-duration frames. The remaining features, primarily concerned with rendering frames and process management, have a significantly lower coefficient, which suggests that they are less likely to be strongly correlated with frame

Table 6.2. Highest-magnitude coefficients for platform-level performance bug workload. Features chosen by feature selection are emphasized in bold.

Coefficient	Description
1.251	**Queue function duration**
-0.360	Grafika thread duration
0.263	mpdecision thread existence
-0.226	Log handler thread duration
0.222	Qualcomm sensor thread duration

rendering time. The decision tree feature selection implementation identified an identical set of features, suggesting some generalizability of the chosen feature selector.

6.4.2. User Interaction

On most modern smart devices, the touchscreen is the primary interface for users providing input to the device. Unfortunately, these touch events take a significant amount of time for the operating system to interpret. Furthermore, touch events typically are connected to some event callback which triggers additional processing, such as rendering animations, or causing some system or network requests. If these callbacks cause contention on the system, it can lead to degraded graphics performance that is visible as dropped frames.

We design a workload to highlight this category of performance bug. We create an application with a button that has an on-click handler that triggers a significant CPU computation. The application also contains an active animation. Because this CPU computation causes contention with the available processing resources on the device, we

Table 6.3. Highest-magnitude coefficients for touch input event example. Features chosen by feature selection are emphasized in bold.

Coefficient	Value
1.072	**Touch event count**
1.049	**Debug log writer thread duration**
0.752	Qualcomm sensors thread existence
0.609	mpdecision thread existence
0.580	System server thread duration

note that there is a certain amount of jitter that occurs when the button's click handler is triggered. We begin gathering a 10-second long trace from `atrace` and manually trigger the on-click handler during `atrace`'s execution.

In both the SVM and the decision tree analysis, the touch event count feature was accurately found as being the primary feature contributing to long frames, as shown in Table 6.3. In the SVM analysis, the second-highest feature is from the debug logging thread; because Android logs data during certain situations when frames are dropped, it ended up being strongly correlated as well. However, it's unlikely that this *caused* frames be dropped, and was rather a result of it. The remaining features are again related to handling processes and system data, and are less likely to be implicated in dropped frames.

6.4.3. Long Operation in Main Thread

Android has a structured system for organizing the process threads within user-facing applications. Every Android application has a so-called UI thread that is tasked with computing any animation and draw requests that the application creates, as well as handling user touch inputs. Furthermore, this UI thread performs any additional computation that the application programmer includes in the application. These events are all processed from an event queue that the UI thread maintains. However, since this event queue is shared by all the associated actors in the application, if one of them runs overly long, the system can become unresponsive because it is unable to handle any further input or draw events. The "correct" way to work around this problem is by offloading any long or indeterminate length calculations (such as database or network accesses) into their own subthread, allowing the UI thread to process events normally. Improper threading is a well-known problem in Android, but is unfortunately still one of the most common performance-related bugs [69].

To test our ability to localize this class of bug, we created an application which displays a simple animation on-screen. Additionally, we intermittently trigger a network access (a ping request to the internet) in the UI thread. These ping commands have varying latencies, typically between 10ms and 60ms; because these pings are of intermittent length, they don't always cause dropped frames, but they do have the capacity to. We collect data from `atrace` for 20 seconds while the animation runs, and then analyze the resulting trace with our system.

The output from our analysis system is presented in Table 6.4, and the selected features were again identical between the decision tree and SVM feature selection implementation. These results are interesting because there are many features that are found to have similarly high coefficient magnitudes. In our workload, the long network access in the ping

Table 6.4. Highest-magnitude coefficients for Android Not Responsive (ANR) bug scenario. Features chosen by feature selection are emphasized in bold.

Coefficient	Value
0.211	**Ping thread existence**
0.211	**Process migration thread existence**
0.199	**Ping thread duration**
0.163	Process migration thread existence
0.152	Log handler thread duration

thread was designed to cause frames to drop, and so we expected this to be the selected coefficient.

However, the ksoftirqd and process migration threads were both also found with significant coefficients. ksoftirqd is a Linux construct which aids in handling a significant batch burst of soft interrupts; this improves performance by reducing expensive context switches. The migration thread is used to aid in moving processes between the various CPU cores. These coefficients give some interesting context to the system state during the `bad` frames; it's likely that the ping network accesses are causing thread migration and a significant number of soft IRQs. Although these threads might not be directly causing the dropped frames, they help debugging problematic system states on the system in these situations.

6.4.4. Memory Bandwidth Contention

The bugs that we have been focusing on thus far have been primarily concerned with "internal bugs" — that is, bugs that are primarily a result of the application or operating system's design decisions. A major differentiating point in this work is its ability to also localize bugs which result from system or hardware contention. This class of bugs is a particular nuisance not only because of their complexity, but also because they can manifest differently depending on the hardware or operating system.

Specifically, one class of performance bug which can be particularly elusive are those which arise from memory bandwidth contention. Computationally heavy algorithms can be particularly hard for mobile processors, as they typically contain smaller caches, smaller RAM, and slower bus speeds due to size and power constraints [16]. When an algorithm runs into contention with the memory bandwidth, the decreased performance can lead to noticeable frame rate reductions.

To gauge our ability to detect memory contention states, we devise a workload which causes low performance due to memory activity. We use a basic animation which contains a constantly moving box, which has corresponding computation required to render the animation. Additionally, we create a background service which periodically allocates and deallocates a significant amount of data from memory. This memory and CPU usage causes significant contention on the operating system, and results in frames being dropped during this memory activity.

These results, presented in Table 6.5 have some slightly different characteristics than the previous workloads. First of all, the SVM and decision trees actually selected different sets of features. Both methods selected memory page allocation/deallocation, but the

Table 6.5. Highest-magnitude coefficients for memory contention-related performance bug example. Features chosen by feature selection are emphasized in bold.

Coefficient	Value
0.255	**MC thread existence**
0.226	**Migration thread existence**
0.195	**Memory page allocation/deallocation count**
0.169	netd thread existence
0.169	thermald thread existence

other selected features differed. The decision tree implementation selected the kernel thread and kernel binder runtimes as features, while SVM selected migration and MC thread runtimes. All of the selected features are all closely related to kernel thread and memory activity, but there is some variance in which features were selected between the two algorithms.

Furthermore, in the SVM implementation, the memory page allocation/deallocation feature, which we expected to be the most significant feature, actually had lower magnitude than the migration and ksoftirqd thread runtimes (although all three were selected as important features after feature selection). This suggests the memory allocations were also causing significant soft IRQ and thread migration activity in the `bad` frames. It doesn't mean that our system was necessarily *wrong*, however; the way that the data was separated indicated that the other features had a larger separating margin. Human understanding of the features is still important when identifying the resulting bug sources.

6.4.5. Processor Governor Bug

Dynamic voltage and frequency scaling (DVFS) [102] is ubiquitous on modern general purpose processors. In Android, DVFS is controlled by the kernel's CPU frequency governor. Because the power savings from a lowered frequency can be quite significant, and mobile devices are significantly battery constrained, Android's Linux kernel is commonly shipped with multiple available governors. Each governor has its own method of controlling the processor frequency. For example, the OnDemand governor quickly increases (or decreases) frequency in response to high (or low) CPU utilization, while the Conservative governor reacts more gradually to utilization levels. With sufficient privileges, these governors can be selected by the user during runtime to change how the DVFS system behaves. However, reducing the frequency has a significant impact on the processor's computational speed.

It is important for our system to be capable of detecting performance problems related to the CPU frequency governor. We created a simple animation which has a CPU-bound calculation to render each frame. Hence, if the CPU's calculation rate drops due to a governor decision, we would expect that the frame rate would drop to some degree as well. We collected data for 20 seconds via `atrace`, and partway through the collection process, we change the governor from OnDemand to Conservative. The net effect of changing the governor is that the frequency drops upon changing and causing an appreciable reduction in frame rate.

The extracted coefficients from this trace are presented in Table 6.6. The CPU frequency is selected as being far and away the most significant feature by both SVM and decision tree-based feature selectors. We used swapping governors to cause a brief frequency reduction, but that is not the only situation that our system can help localize.

Table 6.6. Highest-magnitude coefficients for CPU frequency governor performance bug workload. Note, CPU frequency's coefficient is negative because it is inversely proportional with performance. Features chosen by feature selection are emphasized in bold.

Coefficient	Value
-2.001	Average CPU frequency
-0.077	sh thread duration
-0.070	Log handler thread duration
0.034	Log handler thread existence
-0.033	surfaceflinger thread duration

Because we are including the average frequency for each frame as features, we can also identify performance problems due to poorly chosen frequencies within a single governor. Other system-level CPU frequency decisions can be detected as well, such as those due to thermal throttling.

6.4.6. Vendor Processor Controller Bug

In our early phases of testing and validating the ability of our system to locate system bugs, one of the workloads we tested against was video playback in the native YouTube application. However, in these traces, we noticed that the frames would typically be displayed at a constant frame rate, but would experience somewhat common jitters that resulted from a reduced frame rate. These jitters were noticeable not only in our analyzed traces, but were also visually obvious. The traces contained 20 seconds of `atrace` data from a full-screen 24Hz YouTube video, and we analyzed the trace to look into the features that were identified as important.

The SVM model's coefficients are shown in Table 6.7. The decision tree's selected features were mostly similar; it also selected the sensors thread and mpdecision, but actually selected a network transfer thread duration instead of the surfaceflinger thread duration.

Upon looking more into the purpose of these two threads, we began to understand more about how these features might be implicated in YouTube's dropped frames. In addition to the kernel-based governor that all Linux systems come with, some devices also come with a proprietary CPU controller as well. This allows the CPU manufacturer to make finer-grained power or performance decisions than the OS governor would normally allow. However, these decisions often aren't reported to the kernel governor; even if the kernel governor reports that the processor is operating with all cores at maximum frequency, it doesn't necessarily mean that is the case. This can make it more difficult to identify the source of performance problems, as well as making performance vary more between different hardware. Qualcomm implements this with the `mpdecision` daemon [103], which is a closed-source binary which aids the processor in turning cores on or off. In addition to the processor controller thread, the Qualcomm sensor thread is known to cause significant CPU use in certain situations, leading to degraded performance [61, 110].

Unfortunately, both YouTube and the two aforementioned threads are closed source, which means that we were unable to directly fix the performance bug in YouTube. That being said, because both of these features have been shown to have a significant impact on system performance, these threads are likely implicated in the problematic performance

Table 6.7. Highest-magnitude coefficients for vendor processor controller bug on YouTube. Features chosen by feature selection are emphasized in bold.

Coefficient	Value
1.621	**Qualcomm sensors thread existence**
1.417	**mpdecision thread existence**
-1.410	**Surfaceflinger thread duration**
1.150	YouTube thread duration
0.986	superuser daemon thread existence

on YouTube. This initial phase of performance bug localization is invaluable for helping systems developers know where to begin their debugging process.

6.5. Conclusion

In this work, we presented our system for automatic localization of performance bugs on Android. We designed a system for extracting system-level state and partitioning data into labeled instances which reflect state during individual frames. We then were are able to use general causality analysis via standard feature selection algorithms to identify likely sources of performance problems, using minimal manual analysis.

This work focused on graphical frame rate as our targeted metric; frame rate is a clear user-facing metric which has a direct impact on how users perceive the performance of their devices. However, this is not only performance metric that is important in a mobile system. This system-level analysis framework can be extended to analyze any measurable variable, such as input latency, or even power consumption. Between the complexity surrounding modern mobile devices, and the ever-increasing level of interactivity that these devices demand, the difficulty and importance of identifying sources of bad performance to deliver a good user experience is exceptionally high. A system-level method of identifying these classes of user-facing performance problems is one way to aid in improving the computing experience for users.

CHAPTER 7

Related work

7.1. Context-aware system optimization

Biometric sensors are one way of automatically making contextual data about the user available to the system. With this data, the system is able to get a better idea of how the system's state impacts the user, without requiring any direct user intervention. The field of system optimization from biometric data is relatively new, so I also discuss some other areas that biometric data can be used to optimize.

A number of authors have explored leveraging biometric data to directly optimize the system. Shye et al. [116] propose a work which adjusts microprocessor frequency based on data from biometric sensors in an attempt to optimize for user satisfaction, but doesn't build a predictive model for user satisfaction. Picard and Liu [101] proposes a method of using a variety of sensors to reduce the negative impact that interruptive technologies have on users. Krause et al. [75] describe a method of learning user' system preferences from an array of biological sensors.

Mouse and keyboard actions alone are often used to determine user state in human-computer-interaction scenarios. Fox et al. [44] try to determine user satisfaction based on these in the context of web searching. Macaulay [88] uses mouse click frequency as an indicator for user anxiety. These provide a means of providing additional user feedback to the system without requiring specialized hardware.

Considerable work has been put into systems which use sensors to acquire affective data from users in order to enable computers to improve user-facing tasks, rather than optimizing system state. For example, the educational field has looked into using sensors to improve tutoring systems. Kapoor et al. [68] propose a learning system which uses biometric sensors to predict frustration in users that are using the system. Cooper et al. [29] describe a tutoring system which uses multiple sensors to improve the relevancy of questions on a standardized test studying platform by maximizing user engagement. Burleson [17] propose a virtual agent which helps guide the user through the Towers of Hanoi problem; the agent uses sensors to try to empathize with the user's current emotional state. Prendinger and Ishizuka [104] describe another agent-based interface that empathizes with the user through the use of physiological sensors.

Because biometric sensing devices aren't always available, there is a line of research which instead attempts to draw correlation between system performance metrics and computer performance. Endo and Seltzer [38] explore using OS event handling latency, rather than computational throughput, as a measure for performance. Mallik et al. [89] present a system where video frame rates and application response times are also used as a user-perceivable performance metric for the system, as a proxy for user satisfaction. Gupta et al. [50] studied the effects of resource scheduling decisions on directly reported user satisfaction. Work of this nature is important, as it allows for finer-grained system optimization, but is orthogonal to my work on understanding user state from biometric sensors.

Non-biometric contextual data also has significant system optimization potential. Yang et al. [129] propose a system which uses a model which is trained for individual users and considers application context to control laptop frequency to save power.

7.2. User-aware display power optimization

The display is one of the most power-consuming devices in modern personal computing devices. There is a wide array of work which is related to reducing the power consumed by the display. The foundational work for this is that of vision theory: by understanding how the human visual system works, it's possible to begin to understand the various thresholds required to provide a satisfactory experience, and can then use that understanding in an optimization loop with power consumption and hardware complexity. In this section, I first outline some of the important work surrounding vision theory, and then describe various ways that digital displays have been optimized in light of this knowledge.

7.2.1. Vision theory

A huge number of authors have presented work on how the human visual system perceives light and color, and how the brain interprets that information. A summary of the entire domain would be a doctoral thesis in and of itself, so the papers presented in this section just introduce some of the seminal work in the field, or work which is highly relevant to readability and brightness perception. Stevens [119] presents a paper which gives the basis for the thresholds at which humans can notice changes in visual stimuli. This is extended to other sensors beside vision as well, but humans tend to notice changes in stimuli relative to the initial magnitude of the stimuli. In other words, humans will notice a change in brightness of x units more readily in a dark room than a bright one. Fairchild [39] explores how users subjectively perceive lightness/darkness in an image, and finds that users approximately take the average of a scene's brightness, although this varies between users. Owsley and Sloane [98] attempt to build a model for how well users can perceive real-world images, and find that visual acuity actually doesn't improve their model for image perception, and that contrast sensitivity is more important. Adelson [1] analyzes how users perceive light in objects relative to one another (this accounts for many brightness-related optical illusions).

Visual attention, which analyzes what users focus their vision on, and subjective image quality is also important for vision researchers to understand. Itti et al. [62] look to build a saliency-based (how much something stands out) model of visual attention, which allows researchers to predict what a user will look at in a given image. Vaquero et al. [121] use a saliency model to propose a method of *image retargeting*, which is a method of scaling an image without distorting visually important regions. Datta et al. [33] attempt to build a model which predicts how subjectively "good" a photo is. Users are asked to rate a variety of photographs, and then various image metrics are compared to the subjective ratings. Contrast is found to be one of the more important metrics. Peters II and Strickland [100] presents a series of metrics for *image complexity* in a given scene; this is presented for use in the Automatic Target Recognizer domain.

More directly related to display power optimization, there's a significant body of work concerned with how users directly perceive images and fonts on electronic displays. Hill and Scharff [58] explore the impact that various fonts, font sizes, and colors have on readability, from the perspective of reader reaction time. Fujikake et al. [45] perform a similar study which identifies optimal font sizes in the context of car navigation systems, and presents an age-based analysis of this data. Gould et al. [48] compares CRT displays to paper to compare reading speed, and finds that the electronic display can provide similar readability. Bailey et al. [10] performs a similar study where various fonts are analyzed, in an attempt to find the optimal font size for reading speed. Flynn and Badano [43] analyze how much image quality degrades due to the display's optics and light scattering. Kelley et al. [70] proposes metrics to characterize display readability under ambient illumination.

7.2.2. Display optimization

From this foundational work on vision theory, I now present some papers which attempt to directly address the problem of display power consumption while maintaining good visual properties.

Finding methods of altering the display's image to allow the backlight to be dimmed with minimal impact on the user's perception is one common method of reducing LCD backlight power consumption. Choi et al. [22] propose an early work in the mobile field which significantly reduces display power consumption by using variable color depth and backlight dimming. Chang et al. [19] present a work which allows manipulation of the on-screen image to allow the LCD backlight to be reduced while minimally impacting the user's perceived image. Iranli and Pedram [59] extend this work by presenting a method of using tone mapping of the on-screen image, which uses some more complex calculations that allow the backlight to be dimmed while the image itself is brightened. Tone mapping is used to do this in a manner which considers user image perception, and results in a less noticeable change to the user. Bhowmik and Brennan [14] present another work on perceptually equivalent display dimming, as well as power saving systems which use dynamic image interlacing and variable display refresh rates. Anand et al. [6] present another tone mapping scheme that builds upon this work by specifically targeting mobile gaming. Wee and Balan [124] also target mobile gaming, but instead of using tone mapping, selectively dim areas in a mobile gaming application where the user is not expected to spend a significant amount of visual attention. This work is only applicable to OLED systems, as it does not dim the brightness of the whole screen, but merely regions of the screen. Anggorosesar and Kim [7] describe a system which proposes using multiple dimming regions on an LCD (rather than a global screen brightness) to save power. This allows some of the advantages that OLED's regional dimming characteristics have to offer, while still using LCD technology. Shin et al. [113] explore a method of power-saving image compensation for OLED displays, rather than backlit LCD displays. Xiao et al. [127] use a visual saliency model to identify regions that users are most likely to look at, and dim the areas with low saliency.

Some work also attempts to accomplish power reduction by altering the screen content in a way which is readily noticeable to the viewer, but improves the on-screen image's readability characteristics, allowing display brightness to be reduced while maintaining readability. This is most commonly applied to user interfaces or other artificial constructs, rather than to photorealistic images. Ranganathan et al. [106] explore the feasibility of using *energy-aware user interfaces*, which are designed to use high-contrast colors schemes and UI elements, which allow the screen to consume less power without sacrificing readability. Dong and Zhong [36] propose the Chameleon browser, which can automatically change the browser's content to more battery-friendly colors on OLED screens.

Another way of reducing display power is by improving the display's optical characteristics, allowing the display to reduce the brightness of the backlight while maintaining readability. Zhu et al. [134] present a survey of the design characteristics of transflective LCD displays, which use ambient light to increase the screen's effective brightness without using as much additional backlight power. Lee et al. [80] describe an LCD display which uses either an OLED display or a transflective LCD display, depending on how much ambient light is currently available. Asaoka et al. [9] describe a polarizer-free reflective LCD system, which improves sunlit readability. Guterman et al. [51] explore the question of if it's always better to use the brightness screen possible in digital signage, and concludes that overly bright signs are actually less effective to attract attention. Comiskey et al. [27] present an early work on reflective displays, such as those commonly used in eBook readers. These displays allow low-power operation due to not requiring a backlight, but have a slow refresh rate and poor contrast ratio in poorly lit environments.

Brightness is not the only trade-off that can be made to save power on mobile displays. The display frame rate and image quality (quantization) can also be modified to optimize the system's power. Han et al. [55] propose a system which optimizes power during a screen scroll event by using a frame rate which is variable depending on the velocity of the scroll event. McCarthy et al. [90] analyze the subjective impact of quantization and frame rate on video quality simpler video data takes less energy to decode and display. Ou et al. [97] extend this work by proposing a metric which predicts video quality at varying frame rates and quantization levels.

A number of researchers have performed research which explores how individuals interact with their smartphone's batteries. Rahmati et al. [105] study how users interact with their smartphone's power and battery functions from a UX perspective, and proposes methods of improving this interaction. Ferreira et al. [41] present a study on when and how often users charge their smartphone devices, and investigate what factors cause them to charge their devices, which gives valuable insight into what their battery requirements are. Chen et al. [21] explore what factors contribute to smartphone display battery consumption on a variety of devices. [117] [117] present a survey of the power consumption from a number of real mobile device users, and provide a breakdown of power consumption from a variety of phone subsystems via a power model.

There are a number of ways that mobile display subsystems are optimized in ways that are invisible to the user. These optimizations are important to battery-constrained devices, but are more distantly related to my work as they don't exercise a trade-off with readability or user satisfaction. The New Generation Digital Display Interface for Embedded Applications [120] presents the eDP 1.3 standard (embedded DisplayPort), which is a display a subsystem optimization allows panels to display a static image without requiring data from the GPU subsystem for each static frame. Choi et al. [23] propose a method of applying DVFS to an MPEG video decoder by predicting the required to decode a frame within its given time constraint. This optimization could worsen the user experience if a frame deadline is missed, but the authors state a relatively low error rate.

7.3. System performance debugging

Static analysis is one proposed method of improving application performance by directly finding problematic regions in application code. Jin et al. [65] create a static analysis system which is able to find potential causes of performance problems in a number of popular open sources codebases. Liu et al. [87] perform a similar study which instead targets potential performance bugs in a selection of user-facing applications on Android. Bessey et al. [13] give an overview of the design behind Coverity, which is a popular static bug-finding tool. Kwon et al. [78] aim to predict the runtime of various Android applications via automatically instrumented versions of code. In the work by Shivaji et al. [114], the realm of static code bug prediction via machine learning classifiers is explored. Traditionally, there are thousands or more features which are fed into these classifiers; the authors identify a subset of these features which are useful in identifying these bugs. This differs from our work in that they targeted traditional accuracy bugs (not performance bugs) in a static manner.

Profiling is related to static analysis in that it is used to find bugs and performance problems within a single application, but it operates on post-analysis of workloads rather than static code. Graham et al. [49] provide an early work on profiling which profiles workloads via analyzing the runtimes of called routines. Šimunić et al. [123] create a profiler which is specifically concerned with identifying significant sources of energy consumption in computational kernels in embedded devices. Although static analysis and profiling is able to find some sources of system bugs, it differs from my work in that they can't find bugs which are a result of system state or competing processes, nor are they able to take into account the impact of diverse hardware.

Other research is involved with dynamically identifying performance problems in code, rather than just simply understanding program phase duration, as in profilers. Yang et al. [130] instrument Android program code and significantly increase the duration of common problematic functions like database and network access; this makes unresponsive programs more obvious to the developer. The work by Zhang et al. [133] aids in identifying sources of long “user-perceived transactions”, which are the interval between a user input and that input being reflected on the system. Zhang et al. [132] create a system which attempts to find energy consumption bugs, but accomplishes this by identifying data which is fetched but unused by the system. Kim et al. [72] propose a system which analyzes system traces in an attempt to predict sources of program phase latency, but don’t explore this in the context of any user-facing metrics. Xiao et al. [126] present a system which attempts to identify performance bugs which arise from program segments which can be long-running, depending on workload. Identifying performance problem sources in distributed systems is also an important area of research, as evidenced in works such as those by Aguilera et al. [2] and Chen et al. [20]. Dynamic analysis can also take place at higher levels of the software stack, such as in SherLog [131], which draws inferences as to what happened by using run-time logs and automatically comparing those logs with the program’s source code. Although these works are all useful for debugging a specific power or performance problem, none are able to target directly user-facing problems, such as those related to degraded frame rate.

Nistor et al. [96] analyze the sources of and difficulty in diagnosing performance bugs, as compared to functional bugs. They find that although fixing performance bugs isn’t any more likely to introduce functional bugs than when addressing functional bugs, it can be more difficult to identify them due to their transient nature.

CHAPTER 8

Conclusion

When considering the wide diversity in the requirements of users, the devices they use should reflect and leverage this diversity, rather than ignore it. This work proposes using better context awareness of the user to help understand and react to the user's internal and external context. Also, through the use of performance metrics which more closely relate to user preferences, well-informed optimizations between power and performance can be made. Finally, this work proposes the use of user-facing performance bug localization, which allows the computing experience to be optimized and improved even further.

The focus toward a better user experience has become a top design priority for major computer device manufacturers, especially as these devices continue to become more ubiquitous. Recently, user-space software tends to take the spotlight, as far as the user experience goes. However, the hardware and operating system still is relatively unexplored, and have many areas where the experience can be improved.

Improving the cooperation between man and machine is no simple task; this work is but one piece of what will hopefully continue to become a more concentrated focus on truly personal computing. This will require a shift in mindset for many parts of the computing industry, which has traditionally prioritized raw performance over the experience, but is a change that has significant potential benefit for the end user.

References

[1] E. H. Adelson. Perceptual organization and the judgment of brightness. *Science*, 262(5142):2042–2044, 1993.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[3] M. Alpern and N. Ohba. The effect of bleaching and backgrounds on pupil size. *Vision Research*, 12(5):943–951, 1972.

[4] K. Amano and A. Maruoka. On learning monotone boolean functions under the uniform distribution. In *Algorithmic Learning Theory*, pages 57–68. Springer, 2002.

[5] AMD FreeSync Technology. `http://www.amd.com/en-us/innovations/software-technologies/technologies-gaming/freesync#about`, Mar. 2015.

[6] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 57–70. Springer, 2011.

[7] A. Anggorosesar and Y.-J. Kim. Object-based local dimming for LCD systems with LED BLUs. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 315–320. IEEE, 2011.

[8] Arduino Home. `https://www.arduino.cc/`, June 2015.

[9] Y. Asaoka, E. Satoh, K. Deguchi, T. Satoh, K. Minoura, I. Ihara, S. Fujiwara, A. Miyata, Y. Itoh, S. Gyoten, et al. Polarizer-free reflective LCD combined with ultra low-power driving technology. In *SID Symposium Digest of Technical Papers*, pages 395–398. SID, 2009.

[10] Bailey, R. Clear, and S. Berman. Size as a determinant of reading speed. *Journal of the Illuminating Engineering Society (LEUKOS)*, 22(2):102–117, 1992.

[11] BBC News Android Application. `https://play.google.com/store/apps/details?id=bbc.mobile.news.ww`, March 2015.

[12] J. Beatty and B. Lucero-Wagoner. The pupillary system. *Handbook of psychophysiology*, 2:142–162, 2000.

[13] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM (CACM)*, 53 (2):66–75, 2010.

[14] A. K. Bhowmik and R. J. Brennan. System-level display power reduction technologies for portable computing and communications devices. In *Proceedings of the International Conference on Portable Information Devices (PORTABLE)*, pages 1–5. IEEE, 2007.

[15] A. Blum. Online algorithms in machine learning. In *Proceedings of the Dagstuhl Workshop on Online Algorithms*, pages 306–325, 1996.

[16] G. Bournoutian and A. Orailoglu. Miss reduction in embedded processors through dynamic, power-friendly cache design. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*, pages 304–309. ACM/EDA Consortium/IEEE, 2008.

[17] W. Burleson. Advancing a multimodal real-time affective sensing research platform. *New Perspectives on Affect and Learning Technologies*, 3:97–112, 2011.

[18] G. Castellano, S. Villalba, and A. Camurri. Recognising human emotions from body movement and gesture dynamics. In *Affective Computing and Intelligent Interaction*, pages 71–82. Springer Berlin Heidelberg, 2007.

[19] N. Chang, I. Choi, and H. Shim. DLS: dynamic backlight luminance scaling of liquid crystal display. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):837–846, 2004.

[20] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE, 2002.

[21] X. Chen, Y. Chen, Z. Ma, and F. C. Fernandes. How is energy consumed in smartphone display applications? In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 21–26. ACM, 2013.

[22] I. Choi, H. Shim, and N. Chang. Low-power color TFT LCD display for hand-held embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 112–117. IEEE/ACM, 2002.

[23] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 732–737. IEEE/ACM, 2002.

[24] A. Chowdhry. Apples pulls iOS 8.0.1 update due to issues with dropped calls, September 2014. www.forbes.com [Online; posted 24-September-2014].

[25] CIE 145. The correlation of models for vision and visual performance. Technical Report 145:2002, Commission Internationale de l'Eclairage (International Commission on Illumination), 2002.

[26] D. Cline, H. W. Hofstetter, and J. R. Griffin. *Dictionary of visual science*. Butterworth-Heinemann Medical, 4th edition, 1997.

[27] B. Comiskey, J. Albert, H. Yoshizawa, and J. Jacobson. An electrophoretic ink for all-printed reflective electronic displays. *Nature*, 394(6690):253–255, 1998.

[28] K. Connelly and A. Khalil. Towards automatic device configuration in smart environments. In *Proceedings of the Fifth Annual Conference on Ubiquitous Computing (UbiComp)*. ACM, 2003.

[29] D. Cooper, I. Arroyo, and B. Woolf. Actionable affective processing for automatic tutor interventions. *New Perspectives on Affect and Learning Technologies*, 3:127–140, 2011.

[30] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[31] E. Crane and M. Gross. Motion capture and emotion: Affect detection in whole body movement. In *Affective computing and intelligent interaction*, pages 95–101. Springer, 2007.

[32] H. D. Critchley. Book review: electrodermal responses: what happens in the brain. *The Neuroscientist*, 8(2):132–142, 2002.

[33] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Studying aesthetics in photographic images using a computational approach. In *Proceedings of the 9th European Conference on Computer Vision (ECCV)*, pages 288–301. ECCV, 2006.

[34] P. A. Dinda, G. Memik, R. P. Dick, B. Lin, A. Mallik, A. Gupta, and S. Rossoff. The user in experimental computer systems research. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, page 10. ACM, 2007.

[35] T. M. T. Do, J. Blom, and D. Gatica-Perez. Smartphone usage in the wild: a large-scale analysis of applications and context. In *Proceedings of the 13th International Conference on Multimodal Interfaces (ICMI)*, pages 353–360. ACM, 2011.

[36] M. Dong and L. Zhong. Chameleon: A color-adaptive web browser for mobile OLED displays. *IEEE Transactions on Mobile Computing (TMC)*, 11(5):724–738, 2012.

[37] J. F. Duffy, R. E. Kronauer, and C. A. Czeisler. Phase-shifting human circadian rhythms: influence of sleep timing, social contact and light exposure. *The Journal of Physiology*, 495(Pt 1):289–297, 1996.

[38] Y. Endo and M. I. Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 185–199. ACM, 2000.

[39] M. D. Fairchild. A victory for equivalent background — on average. In *Proceedings of the 7th Color and Imaging Conference*, pages 87–92. IS&T, 1999.

[40] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 179–194. ACM, 2010.

[41] D. Ferreira, A. K. Dey, and V. Kostakos. Understanding human-smartphone concerns: a study of battery life. In *Proceedings of the 10th International Conference on Pervasive Computing (Pervasive)*, pages 19–33. IEEE, 2011.

[42] B. Figner and R. O. Murphy. Using skin conductance in judgment and decision making research. *A Handbook of Process Tracing Methods for Decision Research: A Critical Review and User's Guide*, pages 163–184, 2010.

[43] M. J. Flynn and A. Badano. Image quality degradation by light scattering in display devices. *Journal of Digital Imaging (JDI)*, 12(2):50–59, 1999.

[44] S. Fox, K. Karnawat, M. Mydland, S. Dumais, and T. White. Evaluating implicit measures to improve web search. *ACM Transactions on Information Systems (TOIS)*, 23(2):147–168, 2005.

[45] K. Fujikake, S. Hasegawa, M. Omori, H. Takada, and M. Miyao. Readability of character size for car navigation systems. In *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI)*, pages 503–509. ACM, 2007.

[46] G-SYNC Technology Overview. `http://www.geforce.com/hardware/technology/g-sync`, Mar. 2015.

[47] Google Play Store. `https://play.google.com/store/apps/`, March 2015.

[48] J. D. Gould, L. Alfaro, R. Finn, B. Haupt, and A. Minuto. Reading from CRT displays can be as fast as reading from paper. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 29(5):497–517, 1987.

[49] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[50] A. Gupta, B. Lin, and P. A. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC)*, pages 214–224. IEEE, 2004.

[51] P. S. Guterman, K. Fukuda, L. M. Wilcox, and R. S. Allison. Is brighter always better? The effects of display and ambient luminance on preferences for digital signage. In *Society for Information Display Symposium Digest of Technical Papers*, pages 1116–1119. SID, 2010.

[52] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1-3):389–422, 2002.

[53] M. Hall. *Correlation-based feature selection for machine learning.* PhD thesis, The University of Waikato, 1999.

[54] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. In *SIGKDD Explorations*, pages 10–18. ACM, 2009.

[55] H. Han, J. Yu, H. Zhu, Y. Chen, J. Yang, G. Xue, Y. Zhu, and M. Li. E3: Energy-efficient engine for frame rate adaptation on smartphones. In *Proceedings of the 11th Conference on Embedded Networked Sensor Systems (SenSys)*, pages 15:1–15:14. ACM, 2013.

[56] H. Hashemi, M. Khabazkhoob, E. Jafarzadehpur, M. H. Emamian, M. Shariati, and A. Fotouhi. Contrast sensitivity evaluation in a population-based study in Shahroud, Iran. *Ophthalmology*, 119(3):541–546, 2012.

[57] J. Healey. GSR sock: A new e-textile sensor prototype. In *Proceedings of the 15th Annual International Symposium on Wearable Computers (ISWC)*, pages 113–114. ACM, 2011.

[58] A. L. Hill and L. F. V. Scharff. Readability of websites with various foreground/background color combinations, font types and word styles. In *Proceedings*

of 11th National Conference on Undergraduate Research (NCUR), pages 742–746. NCUR, 1997.

[59] A. Iranli and M. Pedram. DTM: dynamic tone mapping for backlight scaling. In *Proceedings of the 42nd Annual Design Automation Conference (DAC)*, pages 612–617. ACM, 2005.

[60] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: Duration predictions and applications to dvfs. *Micro*, 25(5):39–51, 2005.

[61] Issue 67189: sensors.qcom causing Android OS battery drain. `https://code.google.com/p/android/issues/detail?id=67189`, Mar. 2014.

[62] L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 20(11):1254–1259, 1998.

[63] S. Jha and S. A. Seshia. A theory of formal synthesis via inductive learning. *CoRR*, abs/1505.03953, 2015. URL `http://arxiv.org/abs/1505.03953`.

[64] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 215–224. ACM/IEEE, May 2010.

[65] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 77–88. ACM, 2012.

[66] J. Joskowicz and J. C. L. Ardao. Combining the effects of frame rate, bit rate, display size and video content in a parametric video quality model. In *Proceedings*

of the 6th Latin America Networking Conference (LANC), pages 4–11. ACM, 2011.

[67] M. Kalloniatis and C. Luu. *Webvision: The organization of the retina and visual system*, chapter Visual Acuity. Moran Eye Center, 2007.

[68] A. Kapoor, W. Burleson, and R. Picard. Automatic prediction of frustration. *International Journal of Human-Computer Studies*, 65(8):724–736, 2007.

[69] Keeping Your App Responsive. `http://developer.android.com/training/articles/perf-anr.html`, Mar. 2015.

[70] E. F. Kelley, M. Lindfors, and J. Penczek. Display daylight ambient contrast measurement methods and daylight readability. *Journal of the Society for Information Display*, 14(11):1019–1030, 2006.

[71] G. Kelly. Samsung confirms Galaxy S6 has serious memory problems, May 2015. www.forbes.com [Online; posted 1-May-2015].

[72] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu. Introperf: transparent context-sensitive multi-layer performance inference using system stack traces. In *Proceedings of the 2014 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 235–247. ACM, 2014.

[73] R. Kohavi and D. Sommerfield. Feature subset selection using the wrapper method: Overfitting and dynamic search space topology. In *KDD*, pages 192–197, 1995.

[74] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, Netherlands, 2007. IOS Press.

[75] A. Krause, A. Smailagic, and D. P. Siewiorek. Context-aware mobile computing: Learning context-dependent personal preferences from a wearable sensor array. *IEEE Transactions on Mobile Computing (TMC)*, 5(2):113–127, 2006.

[76] B. M. Kudielka, I. S. Federenko, D. H. Hellhammer, and S. Wüst. Morningness and eveningness: the free cortisol rise after awakening in "early birds" and "night owls". *Biological Psychology*, 72(2):141–146, 2006.

[77] A. Kumar, L. Shang, L.-S. Peh, and N. K. Jha. HybDTM: a coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd annual Design Automation Conference (DAC)*, pages 548–553. ACM, 2006.

[78] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 297–308. USENIX, 2013.

[79] Larger Screens and Improved Resolution Drive Growth in Smartphone Displays, According to NPD DisplaySearch. `http://www.prweb.com/releases/2013/6/prweb10850494.htm`, June 2013.

[80] J.-H. Lee, X. Zhu, Y.-H. Lin, W. K. Choi, T.-C. Lin, S.-C. Hsu, H.-Y. Lin, and S.-T. Wu. High ambient-contrast-ratio display using tandem reflective liquid crystal display and organic light-emitting device. *Optics Express*, 13(23):9431–9438, 2005.

[81] P.-M. Lee, W.-H. Tsui, and T.-C. Hsiao. The influence of emotion on keyboard typing: An experimental study using auditory stimuli. *PLoS ONE*, 10(6), 2015.

[82] D. K. Lewis. *Counterfactuals*. Blackwell Publishers, 1973.

[83] Y. Liang, P. Lai, and C. Chiou. An energy conservation dvfs algorithm for the android operating system. *Journal of Convergence*, 1(1), 2010.

[84] B. Lin, A. Mallik, P. A. Dinda, G. Memik, and R. P. Dick. Power reduction through measurement and modeling of users and CPUs: Summary. In *SIGMETRICS Performance Evaluation Review*, volume 35, pages 363–364. ACM, 2007.

[85] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, Apr. 1988. ISSN 0885-6125.

[86] N. Littlestone and M. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212 – 261, 1994.

[87] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 1013–1024. ACM, 2014.

[88] M. Macaulay. The speed of mouse-click as a measure of anxiety during human-computer interaction. *Behaviour and Information Technology (BIT)*, 23(6):427–433, 2004.

[89] A. Mallik, J. Cosgrove, R. Dick, G. Memik, and P. Dinda. PICSEL: Measuring user-percieved performance to control dynamic frequency scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 70–79. ACM, 2008.

[90] J. D. McCarthy, M. A. Sasse, and D. Miras. Sharp or smooth?: Comparing the effects of quantization vs. frame rate for streamed video. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pages 535–542. ACM, 2004.

[91] *Windows Platform Design Notes.* Microsoft, 2002.

[92] J. W. Miller. Study of visual acuity during the ocular pursuit of moving test objects. II. effects of direction of movement, relative movement, and illumination. *Journal of the Optical Society of America (JOSA)*, 48(11):803–806, 1958.

[93] Mobile Eye: Lightweight Tetherless Eye Tracking. `http://a-s-l.com/products/mobileeye.htm`, June 2008.

[94] N. Na, J. Jang, and H.-J. Suk. Dynamics of backlight luminance for using smartphone in dark environment. In *Proceedings of the International Conference on Perception and Cognition in Electronic Media (HVEI)*, pages 90140I–90140I. SPIE, 2014.

[95] Nexus 4 Tech Specs. `https://support.google.com/nexus/answer/2840740?hl=en&ref_topic=3415523`, November 2014.

[96] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246. IEEE, 2013.

[97] Y.-F. Ou, Z. Ma, T. Liu, and Y. Wang. Perceptual quality assessment of video considering both frame rate and quantization artifacts. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(3):286–298, 2011.

[98] C. Owsley and M. E. Sloane. Contrast sensitivity, acuity, and the perception of ‘real-world’ targets. *The British Journal of Ophthalmology (BJO)*, 71(10):791–796, 1987.

[99] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011.

[100] R. A. Peters II and R. N. Strickland. Image complexity metrics for automatic target recognizers. In *Proceedings of the Automatic Target Recognizer System and Technology Conference*, pages 1–17. SPIE, 1990.

[101] R. Picard and K. Liu. Relative subjective count and assessment of interruptive technologies applied to mobile monitoring of stress. *International Journal of Human-Computer Studies*, 65(4):361–375, 2007.

[102] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102. ACM, 2001.

[103] K. Prasad. Energy measurement and optimization of applications on mobile platforms. MS project report, Boston University, 2014.

[104] H. Prendinger and M. Ishizuka. The empathic companion: A character-based interface that addresses users' affective states. *Applied Artificial Intelligence (AAI)*, 19(3-4):267–286, 2005.

[105] A. Rahmati, A. Qian, and L. Zhong. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 265–272. ACM, 2007.

[106] P. Ranganathan, E. Geelhoed, M. Manahan, and K. Nicholas. Energy-aware user interfaces and energy-adaptive displays. *Computer*, 39(3):31–38, 2006.

[107] M. S. Rea and M. J. Ouellette. Relative visual performance: A basis for application. *Lighting Research and Technology (LR&T)*, 23(3):135–144, 1991.

[108] M. Schuchhardt, S. Jha, R. Ayoub, M. Kishinevsky, and G. Memik. CAPED: Context-aware personalized display brightness for mobile devices. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 19:1–19:10. ACM, 2014.

[109] D. Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Pharmacodynamics*, 15(6):657–680, 1987.

[110] Sensors.qcom eating battery! `http://forum.xda-developers.com/showthread.php?p=51438512#post51438512`, Nov. 2013.

[111] S.-s. Seo, A. Kwon, J.-M. Kang, J. Strassner, and J. W.-K. Hong. Pyp: design and implementation of a context-aware configuration manager for smartphones. In *Proceedings of the 1st International Workshop on Smart Mobile Applications (SmartApps)*, 2011.

[112] C. Setz, B. Arnrich, J. Schumm, R. La Marca, G. Tröster, and U. Ehlert. Discriminating stress from cognitive load using a wearable EDA device. *IEEE Transactions on Information Technology in Biomedicine*, 14(2):410–417, 2010.

[113] D. Shin, Y. Kim, N. Chang, and M. Pedram. Dynamic voltage scaling of oled displays. In *Proceedings of the 48th Annual Design Automation Conference (DAC)*, pages 53–58. ACM, 2011.

[114] S. Shivaji, E. J. Whitehead Jr., R. Akella, and S. Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 International Conference on Automated Software Engineering (ASE)*, pages 600–604. IEEE, 2009.

[115] A. Shye, B. Ozisikyilmaz, A. Mallik, G. Memik, P. A. Dinda, R. P. Dick, and A. N. Choudhary. Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 427–438. ACM, 2008.

[116] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. A. Dinda, and R. P. Dick. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*, pages 188–199. IEEE/ACM, 2008.

[117] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture (MICRO)*, pages 168–178. IEEE/ACM, 2009.

[118] A. Shye, B. Scholbrock, G. Memik, and P. A. Dinda. Characterizing and modeling user activity on smartphones: summary. In *SIGMETRICS Performance Evaluation Review*, pages 375–376. ACM, 2010.

[119] S. S. Stevens. To honor fechner and repeal his law. *Science*, 133:80–86, 1961.

[120] The New Generation Digital Display Interface for Embedded Applications. `http://www.vesa.org/wp-content/uploads/2010/12/DisplayPort-DevCon-Presentation-eDP-Dec-2010-v3.pdf`, Dec. 2010.

[121] D. Vaquero, M. Turk, K. Pulli, M. Tico, and N. Gelfand. A survey of image retargeting techniques. *Applications of Digital Image Processing*, 7798:1–15, 2010.

[122] V. Vovk. A game of prediction with expert advice. *Journal of Computer and System Sciences (JCSS)*, 56:153–173, 1997.

[123] T. Šimunić, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS)*, pages 193–198. IEEE, 2000.

[124] T. K. Wee and R. K. Balan. Adaptive display power management for OLED displays. In *Proceedings of the First International Workshop on Mobile Gaming (MobiGames)*, pages 25–30. ACM, 2012.

[125] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2005.

[126] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 90–100. ACM, 2013.

[127] Y. Xiao, K. Irick, V. Narayanan, D. Shin, and N. Chang. Saliency aware display power management. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1203–1208. ACM/IEEE/EDA, 2013.

[128] Y. Xu, M. Lin, H. Lu, G. Cardone, N. Lane, Z. Chen, A. Campbell, and T. Choudhury. Preference, context and communities: a multi-faceted approach to predicting smartphone app usage patterns. In *Proceedings of the 2013 International Symposium on Wearable Computers (ISWC)*, pages 69–76. ACM, 2013.

[129] L. Yang, R. P. Dick, G. Memik, and P. Dinda. HAPPE: Human and application driven frequency scaling for processor power efficiency. *IEEE Transactions on Mobile Computing (TMC)*, 12(8):1546–1557, 2013.

[130] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Proceedings of the International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, pages 1–6. IEEE, 2013.

[131] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. *ACM SIGARCH Computer Architecture News*, 38(1):143–154, 2010.

[132] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang. ADEL: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the 8th International Conference on Hardware/software Co-design and System Synthesis (CODES+ISSS)*, pages 363–372. ACM, 2012.

[133] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: event-based tracing to measure mobile application and platform performance. In *Proceedings of the 9th International Conference on Hardware/software Co-design and System Synthesis (CODES+ISSS)*, pages 1–10. ACM, 2013.

[134] X. Zhu, Z. Ge, T. X. Wu, and S.-T. Wu. Transflective liquid crystal displays. *Journal of Display Technology*, 1(1):15–29, 2005.