Controlling Green Users for a Happier Cloud

Maciej Swiech

Abstract

Client-server architectures have been used for years, and many modern applications rely on this approach to be able to present rich and complex information to users without needlessly encumbering their local device with computation, and also to be able to keep all data in a (more-or-less) centrally available location, as well as allowing for multiple applications to share data.

When an application has many users, the application backend must have systems in place for dealing with variations in traffic. Modern systems have been designed from the perspective of trying to maintain an appropriate amount of capacity at the backend (including "spare" capacity to handle spikes of traffic). Put plainly—these systems are designed to be able to respond to incoming requests with an acceptable latency given an unknown workload.

Moreover, in systems such as these it is assumed that latency at the end-user must be minimized as far as possible, and that any increase will negatively impact user satisfaction, and thus also impact the bottom line of the application provider. In this dissertation, I explore the relationship between increased latency in cloud-backed mobile applications and user satisfaction. Furthermore, I explore the effects of environmental contextualization, that is, leveraging the emerging cultural trend of being environmentally friendly as a moral good, in shaping the perception of satisfaction with performance.

I have designed and conducted several user studies designed to test all of these relationships, and found surprising and novel results through each of them. Through these studies I show that users' tolerance for delay (which I term the delay tolerance envelope is much higher on mobile platforms than generally believed. Additionally, users are not willing to change their behavior based on the belief that changes would result in more environmentally sustainable outcomes at datacenter backends, but that even simple group pressuring methods, which allow a user to compare their environmental friendliness to others, can increase their willingness to change.

Keywords

Mobile Applications, User Satisfaction, Traffic Shaping, User Studies, Datacenter Sustainability

This work was made possible by support from the United States National Science Foundation through grants CNS-1265347, and CNS-0720691, the United Stares Department of Energy (DoE) grant DE-SC0005343, by a gift from Intel Corporation, and the Natural Science Foundation of China under grant 61300002.

NORTHWESTERN UNIVERSITY

Controlling Green Users for A Happier Cloud

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Maciej Święch

EVANSTON, ILLINOIS

December 2016

Abstract

Controlling Green Users

for

A Happier Cloud

Maciej Święch

Client-server architectures have been used for years, and many modern applications rely on this approach to be able to present rich and complex information to users without needlessly encumbering their local device with computation, and also to be able to keep all data in a (more-or-less) centrally available location, as well as allowing for multiple applications to share data.

When an application has many users, the application backend must have systems in place for dealing with variations in traffic. Modern systems have been designed from the perspective of trying to maintain an appropriate amount of capacity at the backend (including "spare" capacity to handle spikes of traffic). Put plainly—these systems are designed to be able to respond to incoming requests with an acceptable latency given an unknown workload.

Moreover, in systems such as these it is assumed that latency at the end-user must be minimized as far as possible, and that any increase will negatively impact user satisfaction, and thus also impact the bottom line of the application provider. In this dissertation, I explore the relationship between increased latency in cloud-backed mobile applications and user satisfaction. Furthermore, I explore the effects of environmental contextualization, that is, leveraging the emerging cultural trend of being environmentally friendly as a moral good, in shaping the perception of satisfaction with performance.

I have designed and conducted several user studies designed to test all of these relationships, and found surprising and novel results through each of them. Through these studies I show that users' tolerance for delay (which I term the *delay tolerance envelope* is much higher on mobile platforms than generally believed. Additionally, users are not willing to change their behavior based on the belief that changes would result in more environmentally sustainable outcomes at datacenter backends, but that even simple group pressuring methods, which allow a user to compare their environmental friendliness to others, can increase their willingness to change.

Thesis Committee

Peter A. Dinda, Northwestern University, Committee Chair

Gokhan Memik, Northwestern University, Committee Member

Aleksander Kuzmanovic, Northwestern University, Committee Member

Robert Dick, University of Michigan, External Committee Member

Acknowledgments

There have been many people along the way who have helped, supported, inspired, taught, encouraged, and in general driven me to finishing this dissertation. It truly takes a village.

My adviser, Peter Dinda, without whom I would not have embarked on the crazy adventure that is graduate school. It has been an honor and a privilege to work alongside him. I cannot even begin to describe the amount of inspiration and support he has provided over the years, and it is in large part through his tutelage that I was able to take part in such varied and fascinating research. He never shied away from opening an AMD architecture manual to help with whatever questions I had, and knew the right amount of prompting to keep me on track. I would also like to thank him for the teaching experience I received. I feel truly honored to be finishing my Ph.D. work under him.

I would also like to thank my committee members and the various faculty with whom I have interacted with over the years. Prof Kuzmanovic who has helped guide my work since I first proposed it, Profs Memik and Dick who have been great sounding boards since I first started graduate school, Prof Henschen for always being willing to lend an ear or dad joke, and Prof Findler for being on my qualifying committee and always willing to help out around the halls of Ford or with random questions.

I would like to thank my collaborator, Huaqian Cai, as well as his adviser Gang Huang, and all of the hard work and dedication that was put in by them. It was an exciting and fruitful venture to host Cai at Northwestern University, as well as all of the continued collaborations that followed and enabled large parts of my doctoral work. I hope that this collaborative spirit will continue.

I would dearly like to extend my thanks to all of the fine graduate students and members of the larger Northwestern community—my long time office-mates Marcel Flores and Kyle Hale, without whom I surely would have gone insane[1], my racquetball partners and friends John Otto and Zach Bischof, Prem Seetharaman in whom I found one of the best roommates and friends that a busy doctoral student could ask for, and all of the other students and members of the community.

I would like to thank both of my parents, who over the years have not only sacrificed so much in their own lives, but shown me by example what it means to be successful in life, in academics, and also to have fun along the way. They have supported and enabled me every step of the way, and I will never be able to say or do enough to thank them.

But perhaps most of all, I would like to thank my wonderful partner Allyson Westling, who throughout many years has also been an advocate for my success, my career development, and personal development. She has stood by me no matter what, and always believed in my potential. I wouldn't be the person I am today without her.

There are far too many people to thank and acknowledge, and so for anybody that I've missed, please know that I have appreciated you and your support down the years as well! It's been a blast!

Grant funding for my work was made possible by support from the United States Na-

[1]The success of that may be debatable

tional Science Foundation (NSF) via grants CNS-1265347, and CNS-0720691, the United Stares Department of Energy (DoE) grant DE-SC0005343, by a gift from Intel Corporation, and the Natural Science Foundation of China under grant 61300002.

Maciej Święch

Evanston, Illinois

September 2016

Table of Contents

List of Figures

Chapter 1

Introduction

In many modern user-facing applications, the common model is a client-server one, in which *clients* are the applications running on a device, used by *users*, and served by back-end datacenters, or *servers*. By *datacenter* here I mean a large group of networked computers, typically located in one physical location, and tasked with either storing data or computing responses to incoming requests, often performing computing over the data stored. Client devices request either data, whether that be raw data such as images or text, or request computation, such as translations of text into another language or finding a person's most related friends. As the applications become more complex, the servers begin to turn more into services, with an ever-increasing amount of processing and storage being pushed onto the backend, in order to be able to guarantee control over storage, minimum performance requirements, and allowing for any data analysis to be done in a central location, while providing ever-more complex features and content to the clients. In practice, this ends up turning into client-side interfaces that are mainly responsible for requesting information from the server-side, and providing users with means of more interesting or relevant request.

As both the cost of serving requests – whether in size of data to be fetched, or in computational complexity of processing the request – increases, and the pool of users creating requests increases, scheduling requests in the datacenter becomes an increasingly harder problem. Datacenters typically have a *Service Level Agreement*, or *SLA*, that specifies an acceptable amount of time in which requests must be satisfied. Given that incoming workloads are unknown *a priori*, and that the SLA must be kept even during unanticipated periods of high request rate, many datacenter management methods end up sacrificing energy efficiency.

1.1 Glossary

In order to make clear the terminology I use throughout this work, I will now define the various terms I use.

- *Delay Tolerance Envelope*: The amount of user interface or network delay that a user can be exposed to without irritation, e.g. if a user's satisfaction with an application remains unchanged for up to 500ms of delay, we would consider that to be their *envelope*.

- *Environmental Friendliness*: Any effect which reduces the environmental impact of a datacenter. In the context of this work, this translates into lowering the energy usage of datacenters.

- *Environmental Prompting*: By using specific language during both user study and task instructions, we aim to connect degradation in performance with environmental friendliness. The goal of this connection is to evaluate the willingness of a user to alter their behavior in order to increase their environmental friendliness. Based on whether or not a study subject received this prompting, we refer to them as being:

  - *Nongreen*: Study subjects who did not receive any environmental prompting are referred to as nongreen subjects.
  - *Green*: Study subjects whose instructions included environmental prompting are referred to as green subjects.

- **Peer Pressure**: By presenting the aggregate performance of a user's peer group to that user, the user's behavior may be changed or influenced, especially if the performance of the group is higher than that of the user.
  - *Trailing Group*: Study subjects for whom the group delay average was shown to consistently be at a high level are referred to as being in the trailing group, as their normal delay setting is likely to trail behind the group.
  - *Leading Group*: Study subjects for whom the group delay average was shown to consistently be at a low level are referred to as being in the leading group, as their normal delay setting is likely to be leading ahead of the group.
- **User Trace Regions**: During some of the user studies, we collect traces of a subject's delay setting, which is the amount of delay we introduce into the application that the subject is completing tasks on. These traces are 10 minutes long, and we refer to them by the following region names:
  - *Familiarization*: The first 2 minutes of the trace. During this time, the delay is set at 50%, and the subject is asked not to change the delay setting.

  - *Considered*: After the first 2 minutes, the subject is allowed to change their delay setting, and is asked to set it as high as they are comfortable with while accomplishing the tasks for the current application. The subject may be environmentally prompted at this point.
  - *Reconsidered*: At 6 minutes into the trace, the subject receives a reminder that they have the ability to control their delay. This reminder does not tell the subject to increase or decrease their delay, nor does it include any environmental prompting, it simply notifies the subject that they still retain control over a portion of their experience.
  - *Total*: This is the total duration of the trace during which the subject is allowed to control their delay setting, which lasts from the beginning of minute 2 to the end of minute 10 of the trace.

- **Peer pressure interface variants**: We worried that presenting users with a peer pressure and control interface at all times might become too distracting, so we created two variants which we refer to as:
  - *Visible*: In this variant the peer pressure interface is visible at all times, and allows direct control from the user at all times.
  - *Hidden*: In this variant, the interface is only visible during inactive times of a user study, which allows users to concentrate on the current task, but also both control their delay and be shown the group aggregate between tasks.

- *Cultural differences*: The potential effect of cultural trends and/or beliefs on the willingness of users of different cultures to alter their behavior to effect more environmentally friendly outcomes.

- *Virtue Signaling*: A method of communicating the "value" or "amount" of behavioral changes a user is currently undergoing to an outside entity, whether that is to a specific group of people or to the outside world at large.

1.2 Mobile applications

In 2013, the Pew Research Group estimated that 56% of Americans owned a smartphone, with adoption steadily rising [21]. By *smartphone* I mean any cellular telecommunications device that has modern features such as a screen with input capabilities, access to the internet (whether over WiFi or cellular network), and an operating system that enables it to run programs or applications made available that extend the capabilities of the device. As these smart devices become more ubiquitous, it is only natural that the client-server model be extended to applications, and indeed in 2013 Apple announced that their application store contained over 800,000 applications [65]. Many services that initially created mobile-friendly versions of their web services have also transitioned to making standalone applications which offer a faster and more consistent user experience, such as the Facebook application.

As an example, consider making a modern user-facing cloud application such as Pinterest more energy or economically efficient. The application consists of a user interface (the *frontend*), for example an app on a smartphone, that communicates with the core application and systems software (the *backend*). The backend runs in the remote datacenters and the network that are the physical underpinnings of the cloud. User interactions with the frontend result in requests to the backend.

1.3 User satisfaction in the loop

This dissertation is framed in the context of the Empathic Systems Project, whose high-level proposition is that by bringing user satisfaction as a global feedback input to resource management in any finite system will result in better resource allocation and usage. A *resource* here is any finite thing being allocated, such as bandwidth, processing time / power, or power, and *users* here simply are any people that are consuming the resource. Previous work has found that by not trying to allocate for the "average" user, and instead letting the user inform the system of their satisfaction with their resource allocation, and apportioning the resource appropriately so as to not annoy any users, will satisfy users, with a lower overall usage of that resource compared to standard techniques.

1.4 Optimizing for the user

As mentioned above, datacenter management is designed to satisfy the agreed upon SLA. The SLA is an easy metric for determining performance that is acceptable for the canonical user, and all changes to datacenter management must not irritate this user. However, there has been work done to suggest that individual user delay tolerance, that is, the amount of interface or network delay that can occur before a user becomes irritated, can vary highly for individual users [122]. Moreover, the work shown in this dissertation illustrates that even aggregate delay tolerances are much higher than generally believed. I show with a proof of concept traffic shaping scheme that meaningful changes can be made to the characteristics of user request flows while staying within the acceptable tolerances of users.

1.5 Workload at the datacenter

Many different methods of datacenter scheduling and load balancing have been proposed in the literature, which are further discussed in Chapter 9, all of which are made necessary by the unknowable nature of incoming request streams. Many factors influence the nature of request streams. Wave-like surges of requests cause by the so-called “slashdot effect”, or more recently viral blog posts and videos, which create a huge instantaneous demand for a resource, or even by live events, mean that datacenters need to be prepared to deal with huge spikes of incoming traffic at any given point.

1.6 Datacenter sustainability

As datacenter-backed applications become more ubiquitous and more complicated, so too must the datacenters expand to be able to service the feature-rich clients they back. As mentioned previously, real-world datacenters tend to handle incoming requests by overprovisioning resources, which leads to wasted energy. In 2007 the EPA estimated that datacenters consumed 61 terawatt-hours of energy in 2006, 1.5% of the total U.S. consumption, that datacenters were the fastest growing electricity consumers, and that an additional 10 power plants would be required for this growth by 2011. For financial and environmental reasons, there is a large amount of interest in approaches that make the datacenter more sustainable.

Datacenters bring up or shut down hardware to accommodate the offered load of the requests from the application’s users. The assumption that offered workloads cannot be changed puts major constraints on the possible policies to drive these mechanisms and what they can do. It is clear that a given offered load combined with uniform performance requirements will place a lower limit on how many machines need to be active, regardless of policy. Less obviously, the properties of the requests, for example the interarrival time distribution, request size distribution, and any correlations will affect the dynamics of the request stream and thus how much headroom (additional active hardware) the policy needs to preserve.

1.7 Environmental prompting

One of the premises of my work is that by contextualizing users of environmental information, that is, making them aware that a slight degradation in performance has meaningful impacts on the environmental friendliness of their behavior, could increase their delay tolerance envelope. We find that, at least at this point in time, and for study populations based in the United States and China, environmental prompting is not a powerful enough force to enact behavioral changes.

1.8 Virtue signaling and peer pressure

One of the issues with environmental prompting is that although it was designed to have users associate degradation of performance with environmental friendliness, perhaps such “virtuous” behavior at an individual level is not enough to effect changes. Perhaps what is needed is for a user to be able to compare, or even broadcast their virtue level to the outside world for everyone else to see. In my work I will show the efficacy of using peer pressure to let users compare their virtue to that of their surrounding group, and lay out ideas for ways that users could signal their virtue to others.

1.9 Thesis

The process of delivering information, computation, and overall experience to end-users of cloud-back mobile applications has many tradeoffs in trying to ensure that the end-user experience is satisfactory for all users while keeping operating costs low. I claim that the tolerance for delay for end-users is higher than generally believed, with delays as high as 750ms not producing a noticeable loss in satisfaction. Moreover, I claim that by prompting users with environmental information, creating an association between delay and environmental friendliness, and allowing users to compare their environmental friendliness to that of others, this delay tolerance can be further expanded. This tolerance gives some headroom for shaping user traffic workloads such that they benefit datacenter management while not irritating users.

1.10 Joint work with Peking University

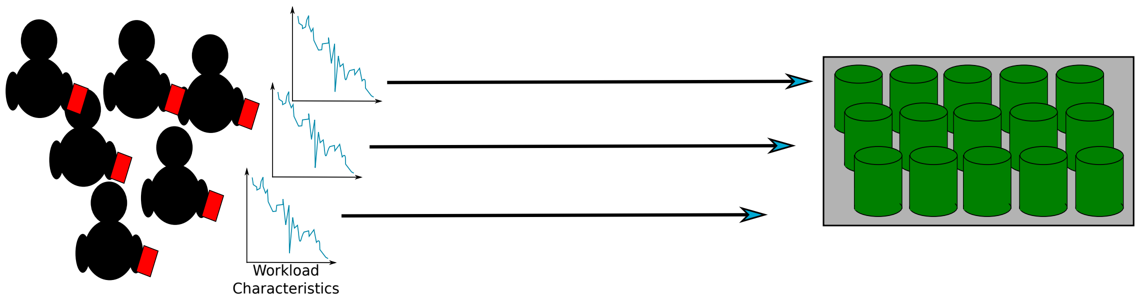
In the process of design and implementing the user studies of Chapters 3,5, 6, and 7 we found ourselves requiring the ability to quickly and easily augment an deploy Android Applications. I was therefore fortunate to have as my collaborator, Huaqian Cai from Peking University, who had helped to develop DPartner—a framework which allowed for automated decomposition of arbitrary Android applications, and facilitated their instrumentation and repacking into applications that could be normally installed on un-altered Android smartphones. The study design would be informed by the ability of DPartner to interpose on features, such as inserting delays into desired API calls, and by its ability to add functionality, such as displaying an overlay interface which could communicate both to and from a central controller portion of the framework. Additionally, 2 user studies were conducted at Peking University, and one using the SoJump crowdsourcing framework, both of which were overseen by Huaqian Cai.

A further outcome of this collaboration has included work done on the DelayDroid system, which allows for the batching of network requests on Android mobile devices, to reduce energy costs by cutting down on the amount of time that the 3G radios in those devices spend in what is called the "tail time", a halfway power state of the radio designed to allow for rapid reconnection to cellular networks. The work for DelayDroid was performed at both Peking University and Northwestern University, and led to a full length publication in the Science China Information Sciences Journal in 2016.

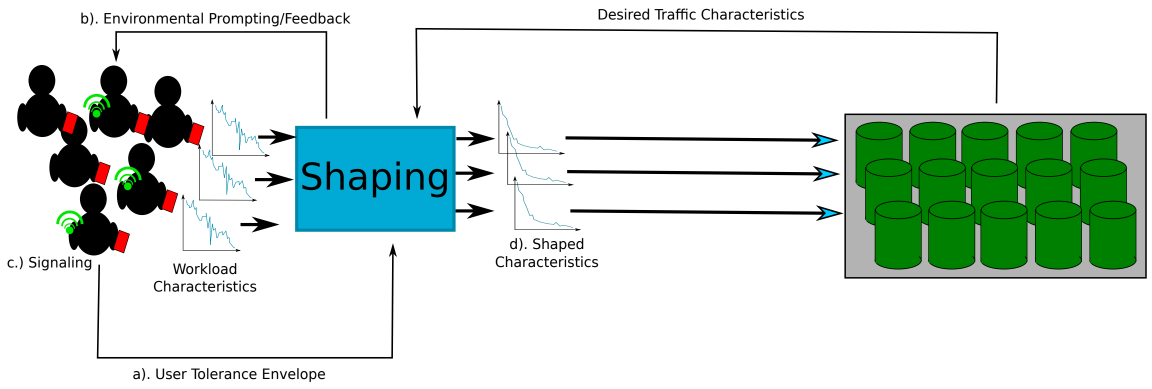
1.11 Outline

Figure 1.1 depicts a view of how interactions between mobile end-users and datacenters occur today, and how I ultimately envision that they could work. The end-users, by nature of using the mobile applications, generate workflows that exhibit a set of characteristics, which are *a priori* unknown. These workflows go into a shaping mechanism, which is informed by 2 things: the user delay tolerance envelope, and the desired workload characteristics from the datacenter. The mechanism does its best to shape the workflows to exhibit the desired characteristics using delay injection while ensuring that these injected delays do not exceed the limit set by the user envelope. The mechanism sends back information to users, which, combined with virtue signaling, attempts to increase the delay tolerance envelop, thus allowing the mechanism to more effectively shape traffic.

I will now describe the overall layout of this dissertation document, and how each piece fits into this ultimate vision. I began by motivating and establishing my thesis statement in Chapter 1, laying out my desire to explore the relationship between delay



Current state of end-user and datacenter interaction.


b). Environmental Prompting/Feedback
Desired Traffic Characteristics
Shaping
c.) Signaling
Workload
Characteristics
d). Shaped
Characteristics
a). User Tolerance Envelope

Envisioned state end-user and datacenter interactions, with traffic shaping that is informed by the delay tolerance envelope, and both environmental feedback and signaling expanding the envelope.

Figure 1.1:

and user satisfaction, and methods for expanding delay tolerance using environmental contextualization and peer pressuring effects. In Chapter 2, I show that by using delaying methods in JavaScript running on mobile device browsers, energy savings can be created on an individual level, while not irritating end users. These findings motivate the need for exploring the potential for such effects on mobile applications, which form the majority of user experience on mobile devices.

With this knowledge, I can now ask the question relating to part **a).** of my vision: what is the size of the delay tolerance envelope? In Chapter 3 I study the delay tolerance envelope of users with an in the wild study, and determine that on average delays of up to 750ms can be added with no significant impact on satisfaction. In Chapter 4 I describe the Kermit Shuffle, an initial approach to shaping user traffic to be more Poisson-like using traces collected from the user study in Chapter 3, and show that we can indeed make meaningful shaping effects while staying with the delay tolerance envelope. This provides one possible mechanism by which portion **d).** of my vision could be implemented.

I now have some evidence about a large delay tolerance envelope among user, and would like to gather additional evidence of this claim, as well as investigate the effects of environmental prompting, as seen in portion **b).** of the vision. Both of these points are covered in Chapter 5 as we replicate the previous study at Peking University, and also begin testing the effects of environmental prompting. The outcomes of this study reinforce the envelope findings of the previous study, and also indicate that, surprisingly, environmental prompting has no effect on the delay tolerance envelope.

We formulate two hypotheses as to the reason for this: 1. that there could be some cultural differences between the study populations and 2. that the lack of any sort of virtue signaling, such as envisioned in portion **c).** of the vision is what is keeping users from altering their behavior. We spend Chapters 6 and 7 discussing user studies that demonstrate that peer pressure mechanisms do indeed produce behavioral changes. In Chapter 8, I discuss potential approaches to interfaces that would leverage the effects of virtue signaling, and would be informed by user satisfaction. I conclude the dissertation in Chapter 10, listing major contributions and potential avenues for future works. Appendices included describe work that I did over the course of my doctoral work that was not directly related to this thesis.

Chapter 2

Can we introduce delays into user-facing applications without annoying users?

In this chapter, I describe the work done on the JSSlow proxy, which demonstrates that there are opportunities for reducing energy consumption of user interfaces on smartphones, and that they can be approached by introducing delays at the user interface level with minimal impact on user-perceived satisfaction. The result of this work was published as a full paper at the MASCOTS conference in 2013.

Modern web sites and web applications include a significant client-side component written in the JavaScript language and interpreted by the browser. The client-side code can manipulate the HTML document via the Document Object Model (DOM), allowing, for example, dynamic page updating, page animations, enhanced controls, and other interactive elements. More generally, JavaScript provides a portable Turing-complete execution model for a high-level scripting language augmented with the capabilities of the browser. The popularity of systems such as Google Documents shows that even complex applications, such as office tools, can be highly effectively implemented in this model.

YouOS demonstrated the extremes the model enables, creating an extensive desktop environment within the model.

Because of the centrality of JavaScript, an important focus of research and development has been on how to execute it faster, including projects such as the Google Closure Compiler [52], which tries to increase JavaScript efficiency and eliminate potential bugs. This is challenging because JavaScript is a dynamically typed language with high-level features, such as `eval`, that generally require interpreted execution. However, it is also important, as the execution model from the perspective of a single page is an event-driven model without threads. While this may simplify application programming because concurrency is not exposed to the developer, it means that a badly-written event handler can block others, degrading the user experience, and in the worst case can result in the browser presenting an "unresponsive script" warning to the user. A well-written event handler running too slowly can result in similar issues.

On a mobile platform, the questions of power, energy, and battery lifetime complicate matters. Slower execution of proper JavaScript may reduce power, and given that internal event generation drives JavaScript execution together with external events, slower execution might also reduce energy and enhance battery lifetime due to less work overall. Finally, slower execution of buggy or peripheral JavaScript might reduce its ability to waste energy and degrade the user experience. Previous work [126] has shown that total JavaScript energy (from transmission and rendering) constitutes a significant portion of energy used during mobile browsing, e.g. 16% on Amazon.com or 20% on YouTube.com.

We claim that on mobile platforms JavaScript interpretation is generally faster than is necessary to maintain a satisfactory user experience, and we propose that JavaScript implementations include a user-configurable throttle. For many sites and users, the throttle can simply be set to a uniform level that will reduce power compared to today's open-throttle setting, while not affecting the user experience. For some sites and users, the throttle is needed so that the user can determine his own trade-off between power and experience.

To evaluate our claims we developed a web proxy system, named JSSlow, that rewrites JavaScript passing through it using the continuation-passing style. Leveraging this, our system introduces what are effectively "sleep" calls into the JavaScript interpretation process, despite the fact that JavaScript has no native sleep functionality. The occurrence of these sleep calls and their arguments constitute the JavaScript throttle mechanism. The throttle is itself set via a global variable.

JSSlow is intended as a proof-of-concept for a JavaScript throttle, and certainly other alternatives, such as changes to the JavaScript interpreter itself, may be simpler. JSSlow does have the benefit of allowing any user direct access to the throttle functionality simply by using our proxy. It also doesn't require any client or server changes, making it easier to study our overall claim.

Using JSSlow, we conducted three studies with an Android mobile phone as the client device. In the first study, we evaluated the power and energy savings that the throttle can provide when faced with buggy JavaScript and advertising JavaScript. Not surprisingly, the benefits are quite substantial. In the second study, we considered the power and energy savings that the throttle, with a default setting, can provide for the top 120 web sites. We found an average power savings of 6%, with some sites showing significantly higher savings. That is, the introduction of JavaScript throttle reduces power across the board and has particularly strong effects for buggy or exploitative JavaScript.

Our final study was a user study in which we had participants come to our lab and carry out tasks using several common web sites/applications on an Android phone, both with and without the JSSlow throttle (with the default setting) active. During the tasks, we measured both articulated user satisfaction and power. We found that for common tasks like commenting on articles or Facebook, there was little difference in satisfaction between the two cases. For fine-grain interactive tasks, such as a game, the impact on user satisfaction varied considerably, suggesting that making the throttle visible in such cases would be preferable.

The primary contributions of this work are as follows.

- We identify an opportunity for reducing mobile device power by throttling JavaScript execution.
- We present the design and implementation of a mechanism for such throttling, the transcoding JSSlow web proxy.
- We show that JavaScript throttling can have a major effect on reducing the power of buggy and advertising JavaScript, reducing these by 52% and 10%, respectively.
- We show that JavaScript throttling, at a default, non-aggressive level, leads to significant power reduction for the top 120 web sites. The average power is reduced by 6%.
- We show that JavaScript throttling, at a default, non-aggressive level, has little effect on user satisfaction for several common tasks. It does have an effect on highly interactive tasks, but the effect appears to be highly user dependent, suggesting that the throttle setting needs to be exposed to the user in such cases.
- We discuss alternative methods for implementing JavaScript throttling.

The chapter is structured in the following way. We begin in Section 2.1 by describing the JavaScript execution model and its uses from the perspective of our claims. Section 2.2 then describes the design and implementation of the JSSlow proxy. Section 2.3 describes the testbed setup that we have used for all three studies. In Section 2.4 we present the outcomes of our studies of buggy JavaScript and of the top-120 sites. Section 2.5 then presents the results of our study of user satisfaction. We then step back to consider the results, describing alternative ways in which a JavaScript throttle could be implemented and presenting recommendations in 2.7. Finally, Sections 3.4 concludes our discussion.

2.1 Why sleepy JavaScript?

JavaScript is a dynamically typed, object-oriented scripting language developed by Brendan Eich. It is implemented by modern web browsers in order to create rich, dynamic web pages. The syntax and semantics of the language are a superset of ECMAScript Edition 3 [37], although each browser implements its own interpreter, and therefore, version of the language. The typical model for JavaScript execution is event-driven, allowing web pages and applications to respond to user input. JavaScript follows the common model of registering event hooks such as `onClick()`, and assigning event handlers to be invoked as a result of events coming in.

In this chapter we examine two popular open-source JavaScript engines - V8, which is written and maintained by Google, as well as SpiderMonkey, which is one of the engines written and maintained by the Mozilla foundation. Since all of the testing we did was done on Android phones using the stock browser, all of our results show the impact of throttling on JavaScript being run by V8.

JavaScript power usage

We argue that JavaScript is being run faster than necessary on a mobile platform, and that it is possible to reduce power with little to no effect on the user experience simply by slowing down this execution. While our focus is on reduced power, we feel that reducing power may also lead to reduction in energy (and thus an increase in battery lifetime).

This may seem counter intuitive at first cut, let us consider the energy of a task to be defined as:

$$Energy = Time_{run} \times Power_{run}$$

This model is commonly believed to exhibit the following property: as the execution rate of the task increases, the power grows and the execution time shrinks, but more importantly, the execution time shrinks at a much faster rate. The implication is that given fixed tasks, the best strategy is to execute them with as high a rate as possible. Computational sprinting [102], where execution rate is raised even beyond sustainable thermal limits for brief periods of time, is the extreme example of this "race to finish" approach.

This analysis makes sense for some workloads and tasks, for example, the image recognition kernel in [102], and perhaps the UI controls on the platform (Section 2.5), but it may not capture the event-driven nature of JavaScript execution for a web page or application. Here, the workload is likely to be continuously arriving, and in many cases, users may not be concerned with the speed at which individual tasks complete, or even if they complete. Most importantly, fast execution of JavaScript workload may well introduce additional tasks, which themselves require more energy to execute.

Consider a user visiting a page. The user will interact with the page for a given interval of time, which we call a *dwell time*. Given a fixed perceived performance of the page, the user's interaction with the page will not change. However, if we lower the execution rate of the JavaScript, we will reduce the power, and reduce the amount of work that needs to be done over the dwell time. Suppose there is an advertising animation on the page. Slowing down the animation's execution will be unlikely to change the user's interaction or satisfaction, but we will have fewer updates to perform over the dwell time, and each update will be done at lower power. Put simply, because the user's perception of the page has not changed, the time spent on the site has not changed either, and thus any reductions in power *directly translate* to reductions in energy. For our argument to hold, we must not alter user behavior, which is why we are so interested in not changing the satisfaction that users have with the websites.

Perhaps more interestingly, we found that energy reductions occurred with the addition of any amount of sleep into the JavaScript code. Our best guess for this is that this allows the JavaScript interpreter to issue a sleep system call, and put the underlying kernel into its idle loop, thus allowing the processor to halt and go into a lower power state. Combining this with the reduction of work done over the same dwell time of the user, we can reduce the power consumption of a browsing session. Given this, our goal is to convince the JavaScript interpreter to slow down in such a way as to reduce power. In other words, we need to convince the JavaScript interpreter to go to sleep.

2.2 JSSlow proxy

The proxy that we have developed, JSSlow, works by examining the body of HTML pages that pass through it, identifying key structures of interest in both embedded JavaScript

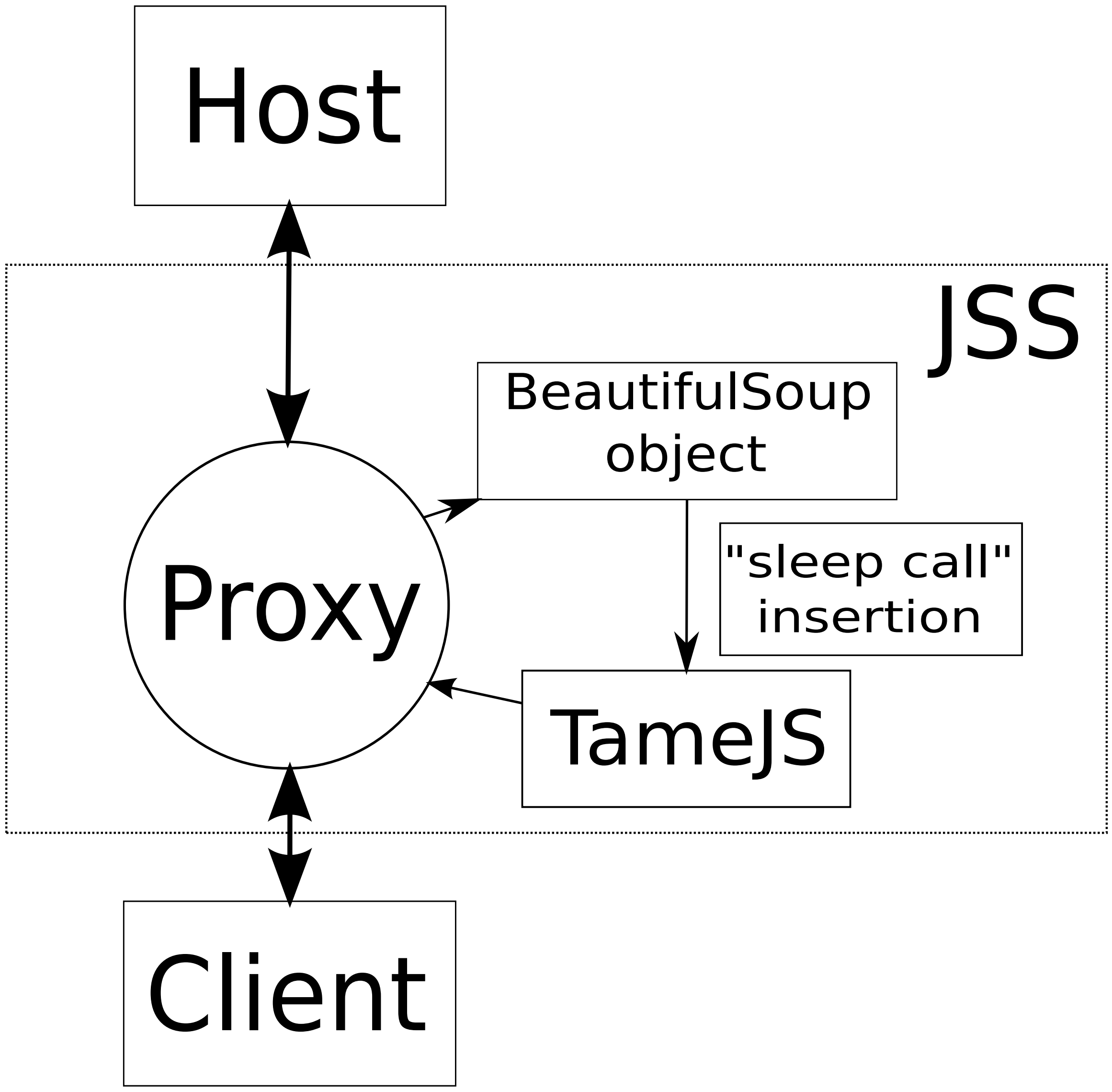


Figure 2.1: JSSlow system design.

code as well as in referenced JavaScript code. Within these structures, for example loop bodies, the proxy inserts the equivalent of calls to a "sleep" function. In this way, we are able to slow down the execution of scripts by ensuring that any code likely to be run repeatedly will have to run our sleep call as well. There were two key difficulties in applying this transformation to scripts, (1) JavaScript does not contain a native sleep call, and (2) the JavaScript code is run a single-threaded context. As we noted in the introduction, this is in keeping with JavaScript's event-based execution model.

Simulating sleep

A sleep call could be simulated in JavaScript's event-based execution model through the use of `setTimeout` timer mechanism, but, as we will show, this approach requires code size comparable to execution steps and thus is unsuitable for looping/recursive code, which is exactly the kind of code we want most to improve. Our adopted approach is to use the continuation passing programming model, explained in Section 2.2, which is supported within JavaScript, albeit not widely used. In essence, our invocations of sleep involve creating a continuation, and then passing that continuation as a timer handler that will be invoked once the desired sleep period has expired. Until the timer event fires, the JavaScript engine can execute independent threads for other pages, and, if it is nothing to do, it will internally await a timer event, using an OS-level `sleep()` or `select()` call. It is the time our page spends in these system calls that reduces the power consumption of the page.

Transforming ordinary JavaScript code into the desired continuation-passing form is challenging. To simplify the process, we leverage TameJS, which has been designed to augment the *server-side* JavaScript language with more straightforward ways of using continuation-passing. By leveraging TameJS on the *client-side*, we can achieve our goal in several steps. First, we parse the JavaScript and identify where we want to inject sleep calls. Next, we inject those calls using the TameJS continuation passing syntax. Finally, we use TameJS to translate back to standards-compliant JavaScript which we then hand to the client. Figure 2.1 illustrates the process.

Continuation passing

Continuation passing, a term coined by Steele and Sussman in their paper on the Scheme programming language [120], is a style of programming in which control flow of a program is explicitly managed with continuations.

The core idea of a continuation is that it captures a complete execution state in such a way that it can be restored at a later time. The continuation is made available at the programmatic level; that is, the program can itself operate on its own continuations, for example explicitly restoring them. Of particular note is the so-called "current continuation", which represents the current execution state. This is often augmented with the notion of "call with current continuation", which allows for control-flow parallelism within the program. As an example, an iterator that operates over tree might be written recursively. The iterator's `next()` function would use a call with current continuation to restore the recursive execution context, make one step on the tree, and then return that value with another call with current continuation back to the caller of `next()`. In this example, continuations allow us to marry the straightforward implementation of a recursive traversal of the tree with in-line iteration over the results of this process. The sharp-eyed reader might note that this process looks a lot like a multithreaded process, and there is in fact an equivalence between cooperative multithreading and continuation-passing.

In the continuation passing style (CPS) of programming, each function takes an additional argument, namely the continuation that the result of the function will be passed to. CPS makes several things more explicit than in “direct style”—the flow of code is immediately obvious since the return of a procedure is directly defined, order of evaluation is explicated since all expressions must be evaluated from the innermost part out, and all calls become tail calls which allows for optimizations.

Since each function must be augmented to include an extra parameter, trying to write code in CPS can be error-prone. This approach is sometimes implemented in compilers such as in Appel and Andrew’s book “Compiling with Continuations” [8] or in TameJS.

TameJS

TameJS [99] is a “source-to-source translator that outputs JavaScript” developed by OK-Cupid as a JavaScript implementation of the C++ Tame framework [75]. It produces standards-compliant JavaScript that can then be run by any JavaScript engine. TameJS was developed with the NodeJS platform [71] in mind. NodeJS attempts to extend JavaScript to the server side, allowing an application developer to build web applications with a single language and execution model. In the server context, especially due to network delays and concurrency among many users, the limited concurrency and cooperative, event-driven execution model of JavaScript can be particularly limiting. TameJS attempts to address these limitations by making the continuation passing style much easier to use within JavaScript code.

TameJS adds two new primitives to JavaScript to facilitate programming with continuations:

- the `await` primitive, which is essentially "call with current continuation", and
- the `defer` primitive, which essentially invokes the continuation that was passed during the call, returning execution state back to the caller.

The combination of `await` and `defer` allow for asynchronous callback code to operate like regular sequential code. The following code illustrates how `await` and `defer` are used to introduce pauses in execution:

```
for (var i = 0; i < 10; i++) {
  await{setTimeout(defer(), 100);}
  console.log ("hello");
}
```

The result of the above code is that "hello" is written to the console 10 times, with a delay of 100ms between each write. The block of code handed to `await` immediately executes, setting a timer that will fire in 100 ms. The `await` call also packs up the current continuation, and passes it to the code. The handler the timer event will invoke consists of the invocation of `defer`. It is not until the `defer` is invoked that the previously packed up continuation is unpacked and restored. At this point, execution continues at the next statement following the `await` statement. Most JavaScript engines will implement the timer delay through `select()` system calls, giving the kernel the opportunity to put the processor in a low-power state if there is no other work to do.

If this code were written without the `await/defer` combination, "hello" would immediately be printed to the console 10 times. Because `setTimeout` is a non-blocking function, the JavaScript engine does not wait for it to return before going on to run the rest of the code. The only way to achieve the desired pause would be to pass `console.log` as the callback function to `setTimeout`. This approach can be applied easily in places where we only want to introduce a single delay, but in order to apply it even to the simple code example above, each iteration of the loop would need to be its own successive callback to a deeper `setTimeout`. That is, we would have to code the 10 iterations of the loop nest separately.

In general, to implement iterated, or nested continuation-passing as in the above example in standard JavaScript would require that we syntactically unroll the iteration or nesting. The beauty of the TameJS `await/defer` extensions is that this is not needed, which allows the implementation of continuation-passing programs in the straightforward style available in other languages that support continuations.

JSSlow

The JSSlow proxy is an extension of a proxy we previously developed to inject JavaScript-based user interfaces that overlay themselves on existing web sites. The initial proxy, which itself was an extension of the Tiny HTTP proxy for Python[1], was designed to support studies of user-centric network scheduling for the uplink of home routers and is described in more detail elsewhere [81]. In this chapter, we generally do not make use of its ability to to inject new user interface components into the user experience, but rather as a framework within which we we can carry out transformations of existing JavaScript code. It implements standard SOCKS proxy semantics and so can be used with any web browser simply by configuring the browser's proxy configuration to use it.

JSSlow observes the HTML body of any response that goes through it, and then analyzes and alters that body before returning it to the requesting client. When it has received the body of the page, it parses it using Python's BeautifulSoup library, resulting

[1] `http://www.oki-osk.jp/esc/python/proxy/`

in a BeautifulSoup object which is an abstract syntax tree (AST). The AST representation and the library then allow us to search and transform the tree. The core operation we do is identify JavaScript code blocks into which we will inject `sleep` invocations.

Sleep macro implementation

Our `sleep` is implemented assuming the existence of TameJS, and takes the following form

```
await { setTimeout(defer(), g_slow); }
```

where `g_slow` is a global variable indicating the sleep duration. We generate a special configuration block to set `g_slow` at page load time. It can be adjusted at any point later, for example via a user interface that can also be injected by the JSSlow proxy. It is important to realize that while `g_slow` is a throttle, it is a very course grain one. Any significant `g_slow` value will typically cause the engine to do a yielding system call, which has a significant effect on power, even if `g_slow` is a quite small, nonzero value. Also note that it is the combination of `g_slow` and the locations at which the `sleep` macro is introduced that constitute the overall throttle. The sleep macro can also be changed to allow for complete deactivation of the sleep functionality, resulting in behavior virtually identical to the unmodified code. We discuss alternative methods of throttling in Section 2.7.

Sleep macro injection

The following pseudocode illustrates the process of transforming the AST to include invocations of our sleep macro:

```
// create AST of the incoming html
html-copy = BeautifulSoup(incoming-html)
```

```
sleep = "await { setTimeout(defer(), g_slow); }"

// iterate over all <script..>..</script> fields
for script in html-copy:
    script-copy = script

    // fetch local scripts
    if script.has_tag("src") && src.is_local():
        script-copy = fetch(src.address)

    insert-at(sleep, "while")
    insert-at(sleep, "if")
    insert-at(sleep, "for")
    insert-at(sleep, "function")

    try:
        script-copy = tame-compile(script-copy)
    except:
        // if compilation failed, just skip
        continue

    script = script-copy

return html-copy
```

In essence, we transform all scripts that are inlined in the page or that are referenced by the page and exist on the same site. Each script has sleep macros introduced into the bodies of its of its control structures (while, if, for, and the entry points of functions). The transformed scripts (including the referenced scripts) are then inlined into the page. If a transformation or inlining of a script fails, we use the original script.

We currently avoid transforming referenced scripts that are not local to the originating page's site. This is because we found that many such scripts would require more subtle transformations in order to be successfully inlined. For example, if a script containing the line `document.write(“</script>”)` were fetched, escaping would be needed to avoid having the string be interpreted by the HTML parser as a script tag. Because of this limitation, we currently under report the potential effect of JavaScript throttling, particularly for throttling advertising and marketing JavaScript, which is almost all non-local. We say more about this missed opportunity in Section 2.6.

Once we have finished adding `sleep` invocations to a script, we run the modified script through the TameJS compiler. TameJS transforms all of our uses of the `sleep` macro back to standard JavaScript, but also includes a `require` call that brings in the TameJS library code. We then replace this with the TameJS library code itself (252 lines of JavaScript), resulting in a script that is as self-contained as the original script.

The transformed script is then inserted back into the body of the page and the proxy continues on to the next script. If any error occurred in the compilation of the transformed script, the original script is instead left on the page. Once all scripts had been processed, the script prepends the initialization code, including the global declaration of the sleep duration.

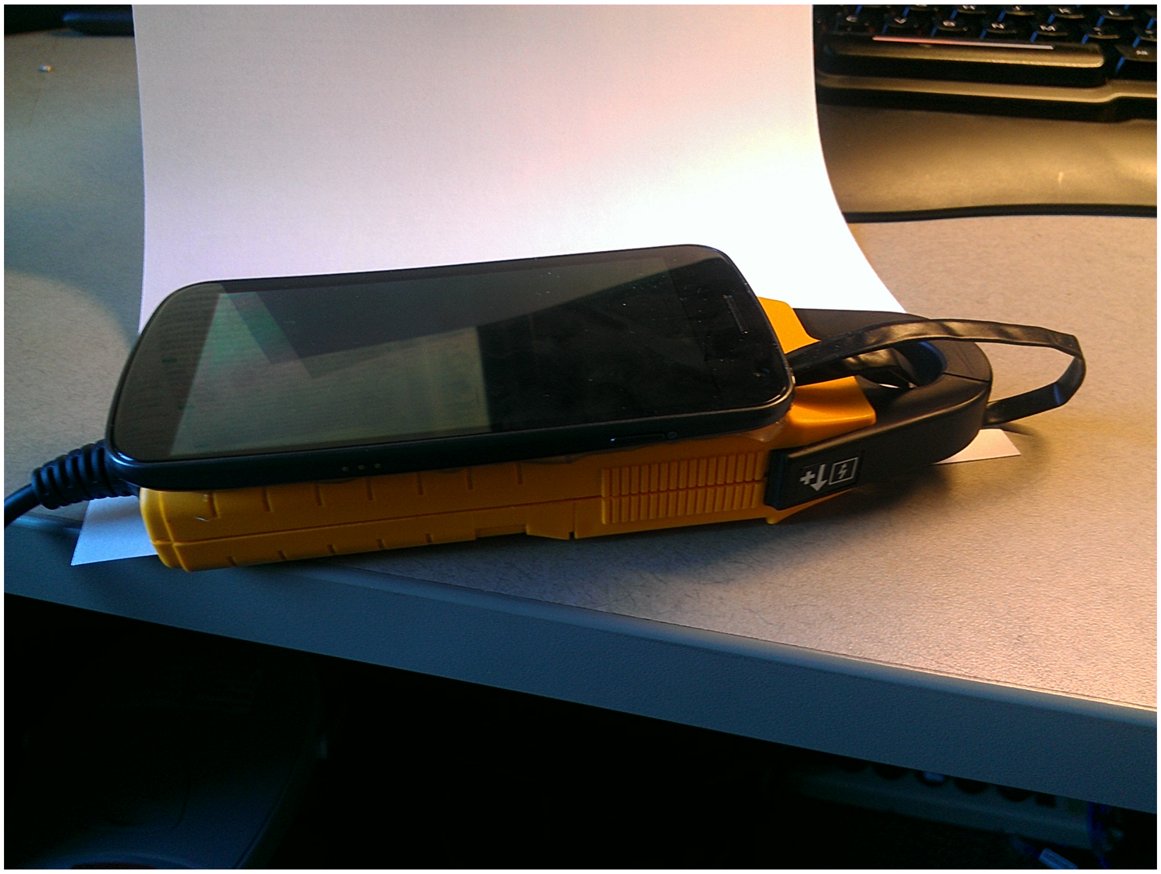


Figure 2.2: Testbed as seen by user.

2.3 Testbed

To carry out the studies described in this chapter, we developed a simple testbed centered around a Google Galaxy Nexus Android phone. The purpose of the testbed is to enable instantaneous power measurement of the phone while allowing the phone to be used either directly by a user or in an automated fashion, following a script. The phone is used on battery power in all cases. The phone is configured to use the JSSlow proxy, which is also part of the testbed. Figure 2.2 shows the appearance of the part of the testbed seen by the user. The box under the phone is a current clamp. The phone uses a WiFi for connectivity, attaching to an access point under our direct control.

Power measurement

Instantaneous power is measured by connecting a conductor directly from the positive terminal of the battery to the positive terminal on the phone, with an insulator directly between the two ends of the conductor, creating a loop external to the phone that current can run through. In order to allow the battery to still fit into the phone (allowing easy connections between the other 3 terminals) a thin conductor was needed. We use layers of metal foil as the conductor.

With the external loop directly accessible, we are able to use a current clamp to measure the current flowing out of the battery. Since this method uses an indirect method of measuring current flow, it has a very low impact on the current itself, which results in accurate readings with little perturbation.

The current clamp we use is a FLUKE i30, which has a maximum current of 20A RMS, and an output of 100 $\frac{mV}{A}$. The device has an accuracy of $\pm 1\%$ at $\pm$2mA and a resolution of $\pm$1mA. The output of the current clamp is fed into a RadioShack 22-812 digital multimeter (DMM), which provides access to readings via an RS232 serial port. This port is wired via a RS232 to USB converter to a monitoring PC. We record DMM readings with QtDMM, software designed for communication with various DMMs. Readings are taken at 10 Hz.

It is important to note that the combination of the phone, current loop, current clamp, etc, remains hand-portable—the user can still hand-hold the phone.

In order to arrive at power, we must multiply the current read from the current clamp by the voltage of the battery. The battery Voltage is subject to some noise, but is relatively stable at 3.8V. This gives us a conversion from voltage read by the current clamp to power of

$$W = V_{clamp} \times \frac{A}{100mV} \times \frac{1000mV}{V} \times 3.8V$$

WebProxyAutomator

Our testbed can be used non-interactively, since it had the ability to both visit and measure the power of a list of sites with and without the proxy enabled. To facilitate such studies, and other non-interactive experimentation, we created an automated testing application, WebProxyAutomator (WPA) for use with the Google Android operating system. WPA creates a `WebView` [51], which is an instance of the built in Android web browser, and then makes successive calls to load a new page from the list at selected intervals, tarrying at each page for a user-selected visit time.

2.4 Opportunity

To evaluate the potential power savings that are available through JavaScript throttling, we used the testbed to study a range of sites without considering user interaction. This work included a study of the effect on buggy JavaScript and advertising, and a study of the effect on the top 120 most visited web sites.

Buggy JavaScript and advertising

The effects of JavaScript throttling on power are most extreme in cases where the JavaScript in question is buggy, typically resulting in an event handler being especially long, or going into an infinite loop. To evaluate this, we crafted an intentionally buggy test site with the following JavaScript,

Scenario	Proxy On [W]	Proxy Off [W]	Diff [%]
Bugs	1.599	3.325	-52%
Ads	1.332	1.472	-10%

Figure 2.3: Power reduction for infinite loop bugs and advertising. Average power over 10 s.

```
var i = 1; while(1) { i *= -1; }
```

which is run before any text is displayed on the page. The result is that the browser becomes unresponsive and the text never appears. When viewing the same page through JSSlow, however, the text immediately renders, the browser is responsive, and the power consumption was less than half that of the untransformed site. This represents the upper bound of power savings that are possible with JavaScript throttling as implemented in JSSlow. This result can be seen in the first row of Figure 2.3. The power is reduced by 52%, which bounds the opportunity for JavaScript throttling.

Advertising makes extensive use of JavaScript, and because ad code is essentially throw-away code, it is more likely to have bugs. To evaluate the effect of JSSlow JavaScript throttling on advertisements, we manually extracted 50 ads from visits to a random subset of 10 sites from the Google's top-1000 sites (more details on this list in Section 2.4) and ran them through JSSlow, the results of which can be seen in the second row of Figure 2.3.

Top 120 most visited sites

To evaluate the effects of JSSlow-based JavaScript throttling in a real world environment, but without user interaction, we used the WebProxyAutomator (WPA, described in Section 2.3) to programmatically load popular web sites with and without JSSlow enabled and measured the difference in power. The specific set of sites we we loaded were se-

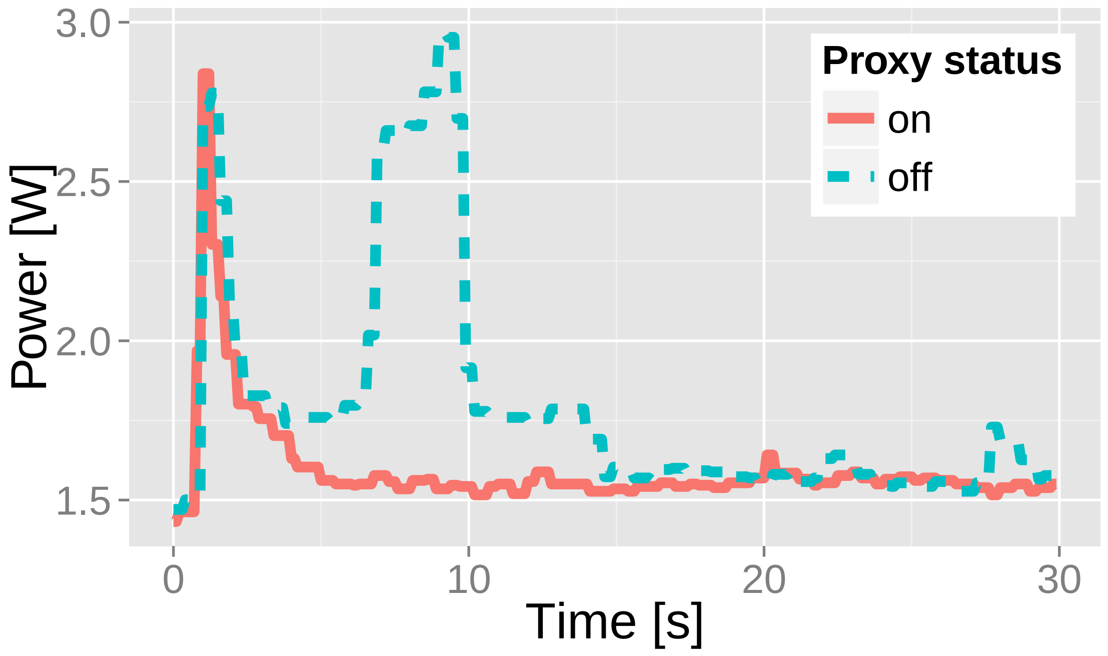


Figure 2.4: Example power signatures with and without JSSlow.

lected from the Google AdPlanner list of the top 1000 sites[2]. Because we ran the phone on battery power, we were only able to visit the top 120 sites before battery depletion.

Each site was visited an average of four times with the proxy on, and four times with the proxy off. A visit included a load and dwell time of 10 seconds, that is, from the time the site started loading to when we advanced to the next site was 10 seconds. The suite of tests was covered in about 2.7 hours, at which point the battery was depleted. During the load and dwell time, the testbed measured the power at 10 Hz. Thus, for one visit, we have 100 measurements. We refer to this as a *power signature*. We can compare power signatures with and without JSSlow, with an example shown in Figure 2.4, but we generally compare averages over the duration of the power signature.

It is important to point out that this automated testing involved no interaction with

[2]`https://www.google.com/adplanner/static/top1000/`

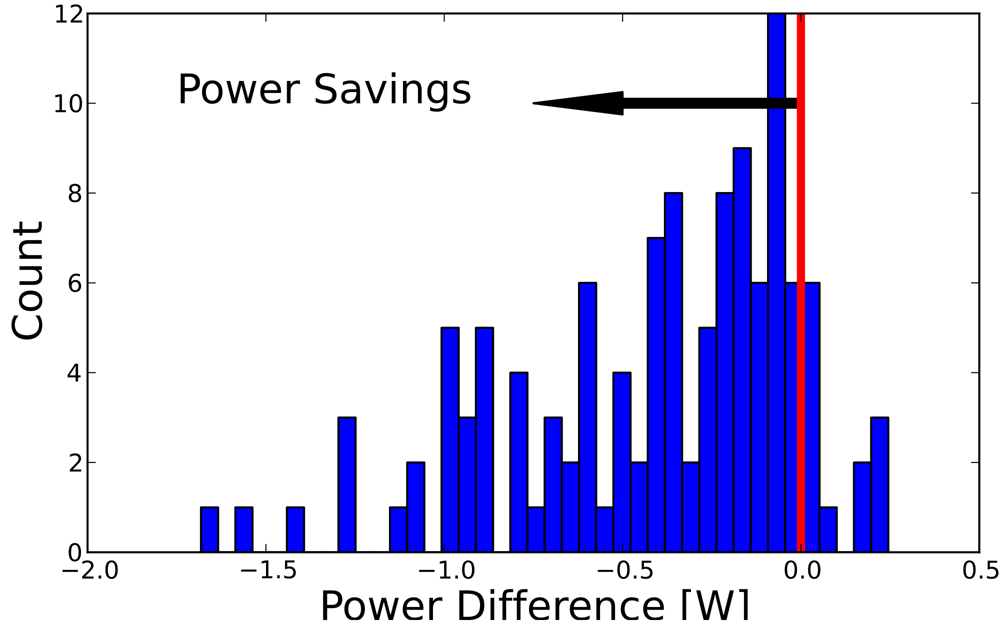


Figure 2.5: Distribution of absolute power difference between JSSlow-throttled sites and native sites. Average power over 10 s.

the contents of the page, thus removing the possibility of running JavaScript from user-interaction driven events such as `onClick()`

Across our sample set, we found an average power reduction of 6% when using JSSlow. However, there is considerable variation across sites. Figure 2.5 shows the distribution of the difference between the throttled and non-throttled runs for the set, while Figure 2.6 plots the absolute power measured for the paired runs. We note that even though there is an overall power savings, there are sites for which we actually increase power. It is important to point out that JSSlow can be disabled in these cases, simply by setting the `g_slow` parameter to zero.

It is important to note that the power savings also include the additional power used to fetch the larger content in the case of the JSSlow transforms being active. For each script, the JSSlow transformation expands the script's size by a factor of two, and adds about

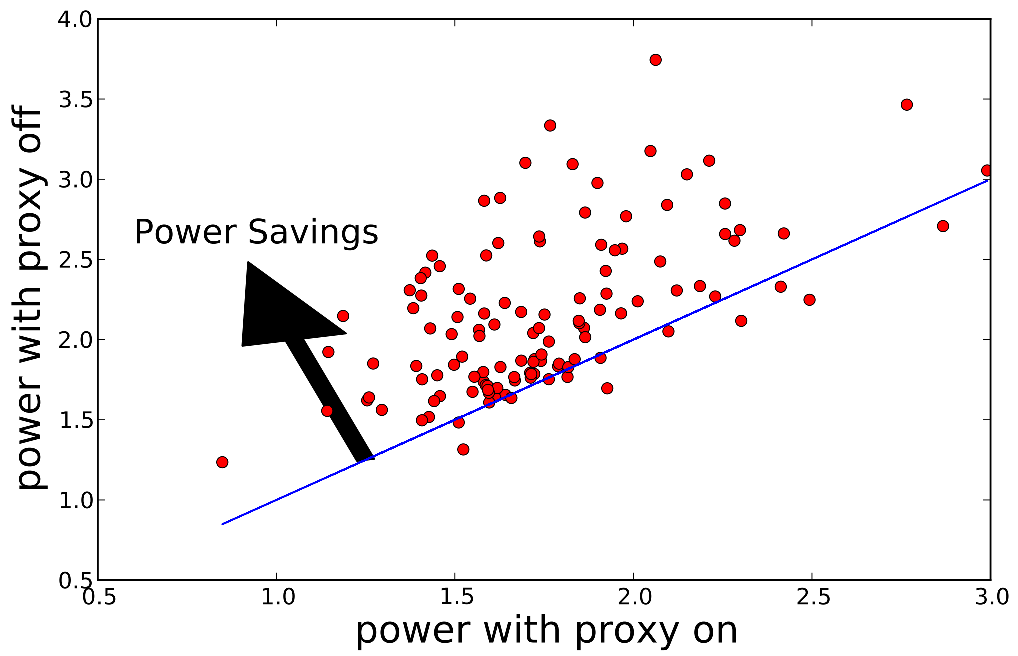


Figure 2.6: Comparison of absolute power with and without the use of JSSlow. Average power over 10 s.

252 lines of library code. In other words, a script of n lines expands to one of $2 \times n + 252$ lines. These results in more energy spent in receiving the script.

As described in Section 2.2, JSSlow does not currently throttle non-local scripts, such as ads. Given that we found that explicitly handling ads lead to power reductions of 10% (Section 2.4), it seems probable that further power reductions will be possible for the top-120 sites once this functionality is complete. Given these results, we believe the baseline average power reduction due to simple inclusion of a JavaScript throttle with a default setting, ignoring interaction, is 6–10%.

Users		
Age	17-25	15
	25-35	4
	35-45	1
Gender	Male	13
	Female	7
Area of Study	Computer Science	10
	Science / Engineering	3
	Liberal Arts	2
	Other	5
Length of smartphone usage	Never	1
	0-1 Months	2
	6 Months - 1 Year	4
	1-2 Years	5
	2+ Years	8
Smartphone Type Owned	None	1
	Android	9
	Blackberry	2
	iOS	7
	Other	1

Figure 2.7: User study demographics.

2.5 User study

JSSlow aims to reduce the power consumption of JavaScript interpretation by slowing it down, which could potentially have an inverse impact on user satisfaction with the web sites running those scripts. We designed and ran a user study that would evaluate the effect of JSSlow throttling on both user-perceived satisfaction, as well as measuring the change in power consumption during real-world usage.

Subjects

Our IRB approval[3] allowed us to advertise our study at various locations throughout the Northwestern University campus. We selected the first 20 students who replied to

[3]Northwestern IRB Project Number STU00002997.

participate in the study. Demographics of this population are presented in Figure 2.7. The study was designed to take one hour, each user was compensated for their time with a $20 gift certificate.

Tasks

The purpose of each task was to approximate the normal usage of a mobile device by a user, as well as to provide various scenarios for which JSSlow might have an effect. The tasks were split broadly among 2 categories: *low interactivity* tasks, and *high interactivity* tasks. What we are trying to capture with these categories is the level of interaction the user will have with JavaScript. The sorts of interaction in low interactivity sites are generally restricted to tasks of the form of leaving a comment, or navigating to a certain part of a site, if the navigation is implemented in JavaScript. High interactivity tasks are ones in which the user is constantly interacting with the script in some way, such as with controls in a game.

CNN and Facebook were selected as low interactivity tasks that are representative of common sites that users visit on their mobile devices. CNN stands in for news and blog-type sites, while Facebook is the exemplar of a social network site.

High interactivity tasks were harder to find, as JavaScript is still relatively new to interactive applications, and browsers do not necessarily implement the same subsets of JavaScript technologies, or with similar performance (Internet Explorer, for example, does not perform well on WebGL benchmarks). Additionally, there are relatively few JavaScript applications that are written to be used with a mobile interface. For these reasons, we chose a very simple application for users—a game of "Snake" written in

HTML5 [4]. The game is controlled by a user swiping the screen in the direction they want the snake to move, with the stipulation that the snake can only ever make 90° turns.

Methodology

We designed an double-blind intervention study in which the subject would either have their code slowed down through the proxy or not, but not be made directly aware of which state they were in. Additionally, the proctor administering the exam would not be aware of the current state either, but only collect information regarding the subject's current satisfaction.

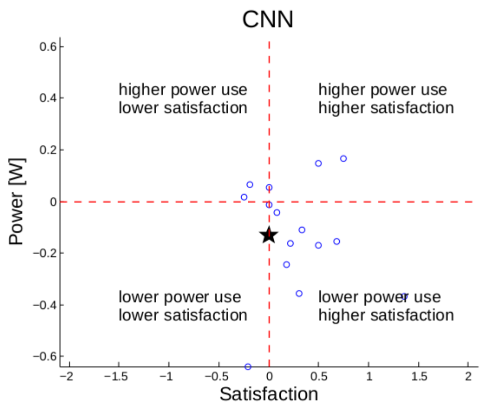
The device that the subject was given to use during the study is the same one described in Section 2.3. We now consider the flow of the study from the perspective of the subject and the perspective of the proctor.

Subject

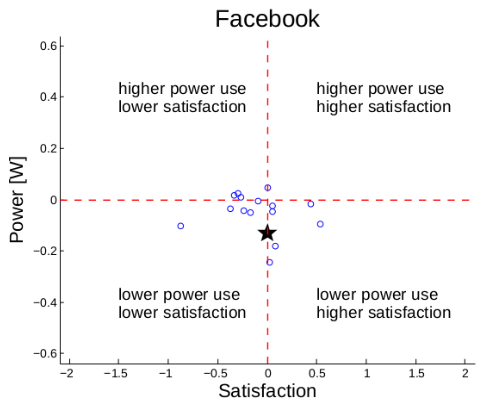
When a subject first arrived, we had them fill out a questionnaire designed to determine their level of knowledge and comfort with a modern mobile device. For the duration of the study, the subject only interacted with web sites in the device's browser, which minimized the amount of experience and knowledge needed of the Android platform.

The user was then given 5–10 minutes to browse to any site. The purpose was to get them familiar with the device, as well as to the normal speeds of the network and browser. The device was connected to the proxy for the entire duration of the study, throttling was turned off during each familiarization phase. Once the user finished familiarizing

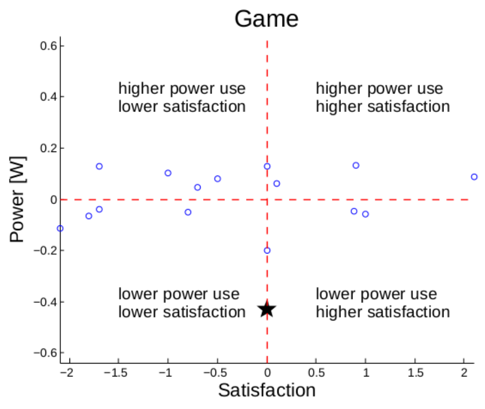
[4] `snake.alexthorpe.com`



(a) CNN



(b) FaceBook



(c) Game

Figure 2.8: Absolute difference in power versus absolute difference in reported satisfaction. Averaged over task intervals. The star represents power difference without interactivity.

themselves with the device, they were prompted to give his current satisfaction with the performance of the device, in order to establish a baseline for that subject.

At this point the device would be pointed to a landing page, which included links to each page the user would visit over the course of the study. For both of the low interactivity sites, a dummy account was already logged in, so as not to require the user logging into a personal account, eliminating any privacy concerns.

For CNN and Facebook, the user would visit an article or page, respectively, read through the content, and then post a short comment. During the course of this process, the user would be periodically prompted to indicate their level of satisfaction with the site. They were asked to verbally express their satisfaction on a Likert scale of 1–10, where 1 represented complete dissatisfaction, and 10 represented complete satisfaction.

Once finished with the low interactivity tasks, the user would be pointed to the high interactivity task, and given 5–10 minutes to familiarize themselves with the speed and reactivity of the game. During this time, the proxy was again off. Once this time was up, the user would play the game for two 5-minute sessions. As in the previous phase, the user was periodically prompted to verbally provide his current level of satisfaction.

When both sessions had been completed, the user was asked to fill out an exit questionnaire, indicting his overall satisfaction with the performance of the device for the entire study. They were also asked to note any times during which they noticed changes in performance.

Proctor and proxy

The study was designed to be double blind—neither the user nor the proctor were aware of the state of the proxy. The proxy had a user-study mode that would expose certain controls to the proctor, allowing it to transition to the appropriate state for each phase of the study. During each interactive phase, the proctor would prompt the user to verbally indicate their satisfaction every 30 seconds, and note this figure down.

The proxy would start in a non-throttling state, so as to allow the user to get accustomed to the baseline behavior during the initial familiarization phase. Once this was done, the proctor would send a signal to the proxy, making it transition to the low interactivity state. During this state, the proxy would randomly choose whether or not to throttle each site that was loaded. The proctor would send signals to the proxy for when the user started each site, allowing careful time stamps to be kept. At the end of this phase, the proctor would send the proxy another signal, putting it back into the familiarization state.

For the first game session the proxy would randomly choose whether or not it would throttle JavaScript, and it would then choose the opposite of that choice for the second run.

During the entire study, the instantaneous power draw was being both measured and recorded. This information, combined with the time-stamped logs and satisfaction results allowed us to extract the average satisfaction for each site in each task, the average power used during the time spent on that site, and whether or not the site had been throttled.

Results

We conducted our study over a total of 20 subjects, and were able to get satisfaction results from 19 subjects, as well as power readings for 15 subjects. We report on these 15 in the following. To account for any anchoring effect due to the user-based interpretation of satisfaction, we did our analysis based on the differences in satisfaction, paired by user.

	Average Difference Off-On			Average	
Task	Avg	StDev	Conf	On	Off
CNN	0.29	0.39	0.12	7.84	7.54
FB	-0.11	0.34	0.11	7.18	7.29
Game	-0.26	1.24	0.39	5.39	5.65

Figure 2.9: Average absolute difference in satisfaction levels with proxy on or off, and average absolute satisfaction levels with proxy on or off.

	Average Difference Off-On [W]			Average [W]	
Task	Avg	StDev	Conf	On	Off
CNN	-0.12	0.22	0.06	2.08	2.20
FB	-0.05	0.08	0.03	1.88	1.92
Game	0.013	0.10	0.036	2.26	2.25

Figure 2.10: Average absolute difference in power (over task interval) with proxy on or off, and average absolute power levels with proxy on or off.

We looked at the values of the absolute differences, as well as the relative differences.

The results of the study are presented in Figures 2.8 through 2.12. Figure 2.8 plots the differences in power versus the differences in satisfaction for each user between slowed down and normal script execution. The figure is split into 3 plots, showing the results for each task. We also show the average power savings in the case of no user input, indicated with a black star.

The tables in Figure 2.9 and Figure 2.10 present the findings in absolute difference. For each task, the satisfaction ratings and power usage for slowed down and normal site performance are compared for that user, and then averaged across all users. We also calculate the standard deviation, and 95% confidence interval for the average value that we found. The last two columns show the average value of satisfaction with the proxy slowing down JavaScript execution across all users, and the average value of satisfaction with JavaScript running normally.

The tables in Figure 2.11 and Figure 2.12 present the same data set, but consider the

Task	Avg [%]	StDev	Conf
CNN	4.24	5.24	1.67
FB	-1.10	5.55	1.81
Game	-0.18	33.10	10.52

Figure 2.11: Relative difference in satisfaction.

Task	Avg [%]	StDev	Conf
CNN	-5.24	10.00	2.92
FB	-2.38	3.61	1.05
Game	0.85	4.55	1.33

Figure 2.12: Relative difference in power.

percentage difference in satisfaction and power usage, rather then the absolute difference. For these values standard deviation, and 95% confidence interval are also calculated.

Analysis

We consider the data from our study using our two broad categories: low-interactivity and high-interactivity tasks. We consider the data from these categories separately since users may have different expectations for how fast each operates. We are testing whether we can proceed power and energy reduction during interactive use, without effecting the satisfaction of the user.

Figure 2.9 is telling us that there is little change in user satisfaction when the proxy is applied in the low interactivity tasks. The confidence intervals suggest that for the Facebook score the detected difference is not meaningful, while for CNN, the detected difference is statistically significant but very small. On the other hand, for the high interactivity task, we see large variation, but with both increased and decreased satisfaction, suggesting that response to performance in high interactivity cases is highly user-dependent. While not statistically significant, the difference between the proxy on and off is quite small. Figure 2.11 tells much the same story. If we average all low interactivity results together, we get a change in satisfaction of 1.6% between the proxy being on versus it being off.

Figure 2.10 shows statistically significant power savings for the low interactivity tasks. Figure 2.12 presents the relative differences. Average all of the low interactivity measurements, we get average power saving of 3.8%. This number is worth comparing with the top-120 study (Section 2.4 where we found a 6% savings, without any user interaction.

For the high interactivity task we find a small increase in power consumption, although it does not rise to statistical significance. Nonetheless, this might seem impossible and contradictory with our automated testing-based finds, but we now consider an explanation.

Impact of the interactive governor

The phone that we used was running Android OS version 4.0.4, which uses the Linux "interactive" CPU frequency governor, whose goal is to provide as smooth a user interface as possible. The governor accomplishes this by ramping up the processor to the maximum power state on any human input, and then evaluates past load to slowly scale down power state [131]. The assumption is that once user interaction has been initiated, more user interaction is likely to happen, and the system should be able to process / respond to that interaction as quickly as possible.

Figure 2.13 shows the instantaneous power draw on the battery in 3 scenarios: (1) a site is loaded by the browser, rendered, and allowed to run for 30 seconds, (2) the same site is loaded and rendered by the browser, but after 25 seconds a user begins typing a comment, and (3) the same site is loaded and rendered by the browser, but after 20

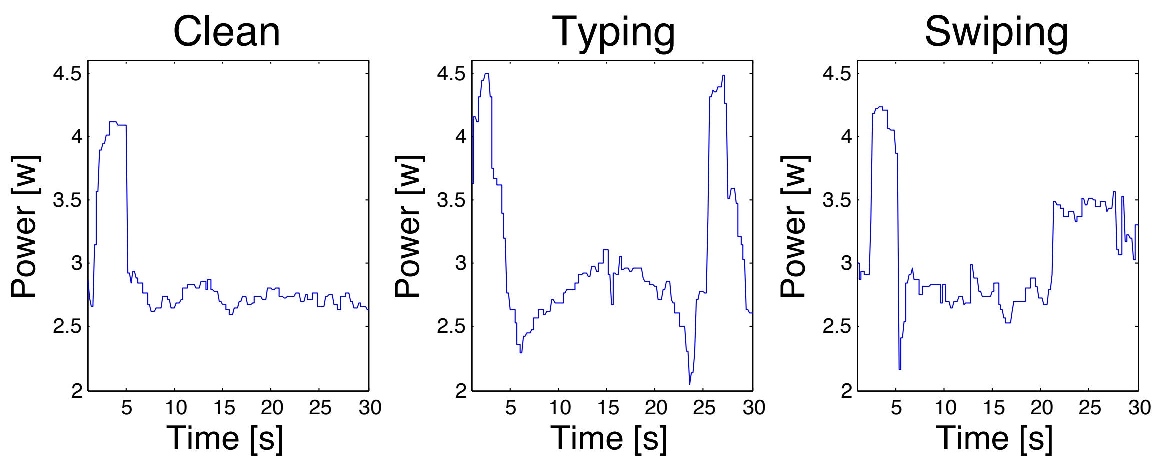


Figure 2.13: Power signature with and without user interaction. "Clean" indicates the site without any user interaction. "Typing" and "Swiping" indicate these activities are occurring.

seconds the user inputs random swipes (continuous contact with the screen). We can see that the power draw during user interaction is on the order of that of the initial page fetch and load in the case of typing, and about half of that in the case of random swipes.

Because of the interactive governor, there will be increased power usage during any time a user is interacting with the device. We claim that the more frustrated a user became with the decreased responsiveness of the game application, the more often they would press the screen, trying to correct errors that had been committed during gameplay because the interface had applied their previous action too late. This effect will also be noticeable in any situation where a user has to scroll through a page often, or has to follow multiple links to get to the content they are looking for.

Power versus satisfaction

In Figure 2.8 we compare the change in satisfaction to the change in power draw for each user, and plot them for each task. We have included dotted lines that separate the space into 4 quadrants. If an increase in power savings came directly at the cost of satisfaction (and vice versa) we would expect clustering of points in the upper-right lower-left quadrants, as a positive difference in satisfaction is good, and a positive change in power usage is bad. These plots include data from all users, places where no data was collected is represented with a difference of 0.

In the CNN results, most of the points are clustered in the lower-right hand of the plot, with a few outliers. This is interesting, in that there were very few users who reported lower satisfaction with slowed down execution. In the Facebook plot, we see a similar distribution of power difference, though at a smaller scale. In this task however, there was much more variance in the change in satisfaction, users reported both positive and negative changes in performance.

This could be partly attributable to the fact that the user had to click through more links to get to the full text of the article. Users were told not to let load time influence their ratings of satisfaction, but they could have taken that factor into consideration anyways.

The plot for the game task has results scattered throughout each quadrant, suggesting that each user's perception of performance is highly varied, and argues for allowing the user to control whether or not a slowdown is applied.

2.6 Limitations of JSSlow

We now consider some of the limitations of our specific implementation of a JavaScript throttle. The JSSlow proxy is a proof of concept, showing that it is possible to slow down execution with no client or server changes, but, as we show in Section 2.7, there may be other, finer grain ways of implementing a throttle that avoid JSSlow's implementation issues.

Continuation passing style

JSSlow's use of CPS to simulate sleep calls introduces limitations related to memory use and performance.

Stack space

Consider a loop. By transforming the loop body to use continuations, we are basically transforming the loop into a recursion. In the worst case, we then would need to allocate additional stack space for each iteration or block of iterations of a loop, as it is a function call. The TameJS compiler tries to limit stack use via tail call optimization, and, strictly speaking, since the original code is a loop, it must be possible to do this optimization. However, the actual optimizer is unable to detect this situation in some cases. In such cases, the runtime stack grows with number of loop iterations resulting in large memory use, and, of course, the script fails if the stack gets too large.

As an example, using the V8 engine included in Node.js, with a 4 KB runtime stack, and a `for` loop with no additional allocation that TameJS was unable to tail call optimize, the transformed script ran out of memory between 13,000 and 14,000 iterations.

Running out of stack space is not an inherent issue with JSSlow's approach, but is a consequence of unrecognized tail call recursion optimization opportunities.

Performance

Another consequence of the JSSlow approach of slicing up program execution via continuations is that there is overhead associated with continuation creation and use. To measure this overhead, we compared normal and transformed versions of a `for` and `while` loop. The body of each loop was a non-blocking arithmetic operation, with no allocation. In the transformed version, each loop iteration became a continuation, but there was no sleep used during the continuation. We ran 13,000 iterations of the for loop (where tail call optimization was not performed) and 100,000 iterations of the while loop (where it was). For each, we measured the time it took for the loop to complete in the base and transformed code. All code was run on the V8 JavaScript engine included in Node.js using a stack size of 4 KB. The transformed `for` loop run times increased by an average of 2.1%, the transformed `while` loop run times increased by an average of 1.6%.

Increased content size

Since JSSlow must not only inject additional "sleep" calls into intercepted code, but also transform the resulting code into code runnable by any JavaScript engine, we are increasing the size of the content being delivered to the mobile device. Given that wireless reception of data also affects battery usage, we must be careful to take this into account, as it will have an effect on our energy savings. Our results do this, as we capture the average power over both the fetch and the dwell time. If throttling were implemented internally in the JavaScript interpreter in the client, however, the additional energy cost due to the increased content size could be avoided.

Energy increases

Our results from the top-120 sites (Section 2.4) indicate that we save energy on most sites by using JSSlow. However, a small fraction of sites actually have increased energy use. Ideally, a tool like JSSlow would be able to deactivate itself for such sites. While we can set the `g_slow` throttle in the JSSlow-transformed code, even when set at zero, the energy costs due to the increased content size and the overhead of continuation-passing execution remain. If throttling were implemented internally in the JavaScript interpreter, however, these cost could be avoided. The interpreter could also adaptively set the equivalent of `g_slow`, testing if a non-zero level actually results in decreased power.

Missed opportunities in advertising

JSSlow currently inlines scripts, but we have difficulty inlining scripts that come from non-local sites. We therefore have disabled this functionality in most of our studies. However, advertising JavaScript is almost always of this non-local form, and, as we showed in Section 2.4, it is particularly amenable to energy reduction by our transformations. The consequence is that we were unable to apply the JSSlow throttle to most advertising, and thus miss this further opportunity to decrease energy use, and possibly increase satisfaction, for typical sites.

Course-grain operation

The JSSlow throttle consists of the locations of the injected sleep invocations, and the duration of the sleep intervals. Even with a very small sleep interval, this can have a quite course-grain effect on the performance/power tradeoff. If we inject the sleep macro in every loop iteration, we have the potential of invoking the kernel sleep system call once per iteration. This would slow the loop to one over the minimum sleep interval the kernel provides. In the case of a typical desktop Linux kernel, configured with a 1 ms periodic timer, this would potentially decrease the loop iteration loop to as little as 1 KHz. Having finer grain control over when and how often the JavaScript interpreter thread yields would allow for a more gradual tradeoff between performance and power.

2.7 Recommendations

Despite its limitations, the JSSlow implementation of the JavaScript throttle, and the results we have derived from using it, are, we believe, sufficient for us to make a set of recommendations about the possible mechanisms and policies of JavaScript throttling.

Deploy JavaScript throttling

JSSlow and the results we have developed using it show that it is feasible to reduce power and energy on mobile devices via JavaScript throttling with little effect on user satisfaction. Arguably, the limitations of JSSlow's implementation of throttling actually understate the case.

JavaScript throttling could potentially be deployed in an incremental way, using a proxy approach, like JSSlow's, to provide throttling for existing, unmodified client software talking to existing, unmodified sites. Over time, more sophisticated forms of throttling could be rolled out.

Throttle the engine

The limitations of JSSlow, described in Section 2.6 could be avoided if the throttle mechanism were embedded into the JavaScript interpreter, or engine, itself. The interpreter main loop could simply decide on whether to yield or sleep on each iteration. This would provide a very fine grain throttle while not changing any JavaScript syntax or semantics. Furthermore, since the throttle would be part of the browser implementation, instead of embedded into the JavaScript code itself, as in JSSlow, there would be no code expansion limitation nor any continuation passing overhead.

We examined two JavaScript engines, V8 and SpiderMonkey, with the goal of evaluating the challenges of such an implementation. In both cases, we were readily able to find good points at which to add throttling.

In the V8 codebase, we added a sleep call to the `Invoke` function, located in the file execution.cc. The `Invoke` function marks a point of entry for execution of code in the V8 virtual machine. Through the use of selective invocation of a `nanosleep()` system call with different parameters, we are able to slow down JavaScript execution with arbitrary precision.

In the SpiderMonkey codebase, we identified the `DO_NEXT_OP` macro, located in the filed jsinterp.cpp, as a location to implement a throttle. `DO_NEXT_OP` is the step in the SpiderMonkey state machine during which the next script opcode is fetched and processed.

To control this mechanism, we could provide programmer-driven policy or a user-driven policy, both of which we describe next.

Expose the throttle to the programmer

Given a throttle built into a JavaScript engine, as described in the previous section, a natural question is what policy should control this mechanism. It is tempting to simply expose the sleep functionality directly to the developer, but this might conflict with JavaScript's event-driven model, and would probably require the introduction of threading to be effective. Calling sleep in part of the code should not put the entire document to sleep. For example, we would not want to stop user elements from rendering while we wait for non-essential code to run again.

One alternative that we believe would be fruitful is to directly expose the control parameter that the engine is using internally to decide when to sleep and for how long. However, this would have the disadvantage of breaking the abstraction between the JavaScript code and any given interpreter—we would like the programmer to be able to adjust the throttle of any interpreter.

Another alternative, which would maintain this abstraction, would be to codify a set of API functions that can be provided by any interpreter, functions that provide for relative throttle changes, and extremes, such as: `throttle_up()`, `throttle_down()`, `throttle_max()`, and `throttle_min()`. Extending control through an API such as this would allow the application developer to provide input into the throttle-setting process in an engine-independent way, while leaving ultimate control to the engine developer. This approach also has the advantage of not requiring any change to the standard JavaScript execution model.

Expose the throttle to the user

In the user study that we conducted, throttling applied to the high interactivity task resulted in a high variance of changes in satisfaction. Considering that each user likely has both a distinct perception of performance change, as well as a different expectation of application performance, this makes sense, and argues for exposing a level of control to the user. In fact, in later chapters we develop interfaces wherein users are allowed to set their own delays and we see a high variance of settings.

In this model, throttling would be turned on by default and set to an initial value that would either be determined by the engine developer, the application developer, or the mechanism we describe in the next section. The user would be presented with a method to change the throttle setting, or turn off throttling altogether, perhaps in the form of a visible button or toggle, or via a physical button, or via a special swipe pattern, or biometrics, or perhaps even via a maneuver that is detectable using an accelerometer. In subsequent studies we give users the ability to control their delay using the volume keys on their smartphone. In real world conditions, we could not simply highjack such necessary keys, but it would not be difficult to imagine adding similar functionality, for example using NFC buttons attached to the phone such as those from DIMPLE.IO [33].

The change in throttle level from the default would be stored locally, so that the user would not have to change this every time they revisited the application. Previous work (e.g. [117, 81, 118, 85, 84]) has shown that by allowing users to set their own level of performance leads to power reductions as well as more satisfied users, and that users are often quite capable of setting such throttles.

Socialize the default throttle setting

A shared problem with any policy for setting the throttle is how to determine the default setting, which will represent a good tradeoff of satisfaction and energy savings. Users or applications could then modify this setting as needed. One approach might be, for each site, to select a response latency bound for controls (e.g., 15 ms as in [23]), and then select a throttle setting that just barely achieves this. The approach we prefer would be to have the JavaScript engines report user settings per site to a common service. This service would integrate the settings to determine average or median settings for a site that it would then convey to users visiting the site for the first time.

2.8 Conclusion

We investigated the claim that throttling the execution rate of client-side JavaScript can lead to lower power and energy on mobile devices without a significant decrease in user satisfaction with the the sites that employ the JavaScript code. We have generally found this to be the case, particularly for lower-interactivity sites. Our studies were done with an implementation of throttling that is based around JavaScript transformation in a web proxy. While this has the advantage of working with any client or site, its limitations may also understate the case. Nonetheless, it appears that even a simple throttle implemented in this way, with a default, unalterable setting, is able to reduce average energy over a range of sites by 4–10%, with lower interactivity leading to more savings. These results recommend the deployment of JavaScript throttling, and we provided a range of recommendations on how to do so.

The results we obtained in the work show in this chapter further support our claims that there exists a variance of delay tolerance satisfaction amongst users, and show that through the introduction of delays on client devices, energy savings can be achieved. The work also shows that delay tolerant varies across application, meaning that there exist classes of applications which could be heavily delayed, without annoying most users.

However, mobile web browsing is but a small part of the whole smartphone experience, and inserting delays into JavaScript misses on the opportunity that exists in native smartphone applications. In the rest of my doctoral work, I focus mainly on native smartphone applications, and the new opportunities that inserting delays there opens up.

Chapter 3

What is the tolerance users have for delay in cloud-backed mobile applications?

In this chapter, I describe work that lays out the concept of a delay tolerance "envelope", and demonstrates, with an in-the-wild user study, that the envelope for users is much larger than one might initially suspect, with an average of 750ms being acceptable based on the study population. The results of this work have been accepted and will be published in the MASCOTS conference in 2016.

Policies for scheduling, mapping, resource allocation/reservation, power management, and similar mechanisms are generally designed with the assumption that the offered workload itself is sacrosanct. Even for closed systems, we hope that the system will have minimal effect on the offered workload. For the present chapter, we consider three parts of this assumption: (1) the workload's statistical properties are a given, (2) the overall offered load is a given, and (3) the performance requirements are uniform across users. The design of a policy typically is then focused on the goal of minimizing cost, power, energy, latency, etc. given these.

In this chapter, we consider the prospects for relaxing the assumption that incoming workloads are immutable. Instead of studying how the cloud or datacenter might respond better to a sacrosanct offered workload, we turn the problem around 180 degrees and consider a model in which the backend determines its desired workload characteristics and the frontend, or load balancer, enforces these characteristics. We think this can be done by taking advantage of the variation in tolerance for a given level of performance that exists among individual users. We have found that such variation exists in many other contexts [54, 91, 125, 81, 94], and that its is possible to take advantage of it in those contexts [85, 84, 86, 87, 90, 115, 138, 78, 81, 124, 123].

We leverage a toolchain that lets us interpose on existing popular Android applications taken directly from the Google Play store. Using this toolchain, we modify a set of such applications so that their frontend/backend interaction passes through code that can selectively delay the interaction. Our additions to the applications also include mechanisms for user feedback about satisfaction with performance. This allowed us to conduct a user study where we introduced varying amounts of delay into the applications' frontend/backend interactions, and collected user feedback about their satisfaction with this added delay. A core outcome of the study is that it is possible to introduce up to 750ms of delay without a change in user satisfaction (to within 10% with $> 95\%$ confidence) for our test applications. We also observed that user satisfaction with specific amounts of delay varied considerably.

The contributions of this chapter are:

- We show, via a user study involving 30 users running 5 popular Android applications on their mobile phones over a period of a week, that there is more tolerance for delay among users than generally believed. The tolerance for introduced delay exists across the whole subject group, and it varies across individuals.

3.1 Frontend augmentation

Our user study is based on popular Android Java applications that are available only in object code form, from the Google Play store. We modify these application frontends to add the following functionality, all within the mobile phone.

1. The ability to introduce delays to the frontend's network requests. Delays are selectively introduced according to test cases loaded onto the phone.
2. Continuous measurement of the phone's environment. This includes CPU load, network characterization (RSSI), which radio is in use, and others.
3. An interface by which the user can supply feedback about performance. A user can do so at any time, but can also be prompted to do so by a test case.

The specific choice of applications, test cases, arrival process for test cases, and users is the basis of our study.

The application augmentation framework we use for this study is based on Dpartner [139] and DelayDroid [64]. Dpartner is a general framework for decomposing a compiled Java application into its constituent parts, adding interposition, instrumentation, and other elements, repartitioning the elements differently (for example, across the client/server boundary), and then reassembling the applications. It is intended to support various kinds of experimentation with existing, binary-only, distributed applications.

DelayDroid leverages Dpartner to augment mobile Android applications. We use DelayDroid to add delaying capabilities in applications. DelayDroid effectively introduces

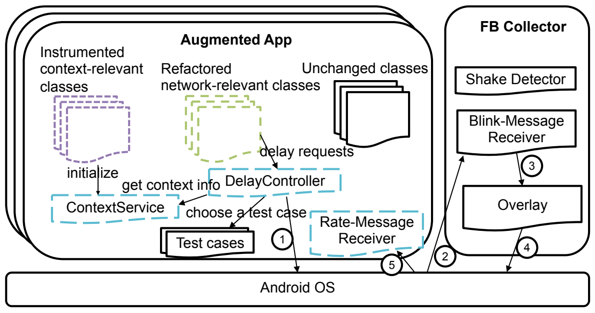


Figure 3.1: Run-time architecture on the frontend.

a proxy into an application through which both high-level (e.g. HTTPWebkit) and low-level (e.g. TCPSocket) network requests are passed. The framework interposes a delaying proxy on 110 networking-related functions from within 5 broad categories of the Android API: HTTPWebkit, HTTPApache, HTTPJaveNet, TCPSocket, and UDPSocket. This allows us to interpose on any network request, irrespective of the method the original app developer chose for their network communications. The proxy can delay requests through any of these interfaces, and is controllable via the test case. In the case of a 0 delay test case, the overhead of the proxy is negligible. The application binary produced as a result of DelayDroid interposition is only about 1-2% larger than the original binary.

In addition to the augmented application, our framework introduces a separate component, the Feedback Collector (FBCollector) that is responsible for coordinating augmented applications on the phone, and collecting user feedback.

Figure 3.1 illustrates the run-time architecture of our system when deployed in a user study. The phone contains multiple augmented applications. For each augmented application, our framework has modified or introduced four kinds of classes: refactored network-relevant classes which we interpose on to send network requests, refactored context-relevant classes which we interpose on to track context, DelayDroid run-time classes which we add in order to inject delay, and unchanged classes.

The DelayDroid run-time consists of 3 components: ContextService, DelayController and Rate-Message Receiver. The ContextService collects and provides information about the context, such as the network status. The DelayController is in charge of injecting delays into the network requests. The DelayController chooses test cases randomly from within a predefined set. Operationally, when any network request occurs, control flow is detoured through the DelayController. The DelayController then delays the request according to the test case. The Rate-Message Receiver interacts with FBCollector, and its operation is explained below.

The FBCollector is a separate application that coordinates with augmented applications by sending and receiving Android broadcast messages. This is also the user-visible portion of the system. A Likert scale overlay of 1-5 hovers over the application, shown in Figure 3.2. The user can select a rating (i.e. rate their satisfaction) at any time. A rating of 1 indicates complete dissatisfaction with performance, and 5 indicates complete satisfaction with it. In addition to unprompted feedback, the user interface also supports prompted feedback, through the form of making the overlay blink a red border.

During the user study, test cases are started at random times by the DelayController. For any test case, a delay is randomly chosen and applied to all instrumented network requests sent over the duration of that test case. Once the test case finishes, the DelayController will send a message to the FBCollector ①, where it is received by the Blink Message Receiver ②, causing the overlay to blink, prompting the user ③. When a user provides feedback via the overlay, the FBCollector notifies ④ the Rate Message Receiver ⑤ component, which in turn logs the feedback and all relevant contextual information.

The log files produced by the system include a timestamp, satisfaction rating, if the

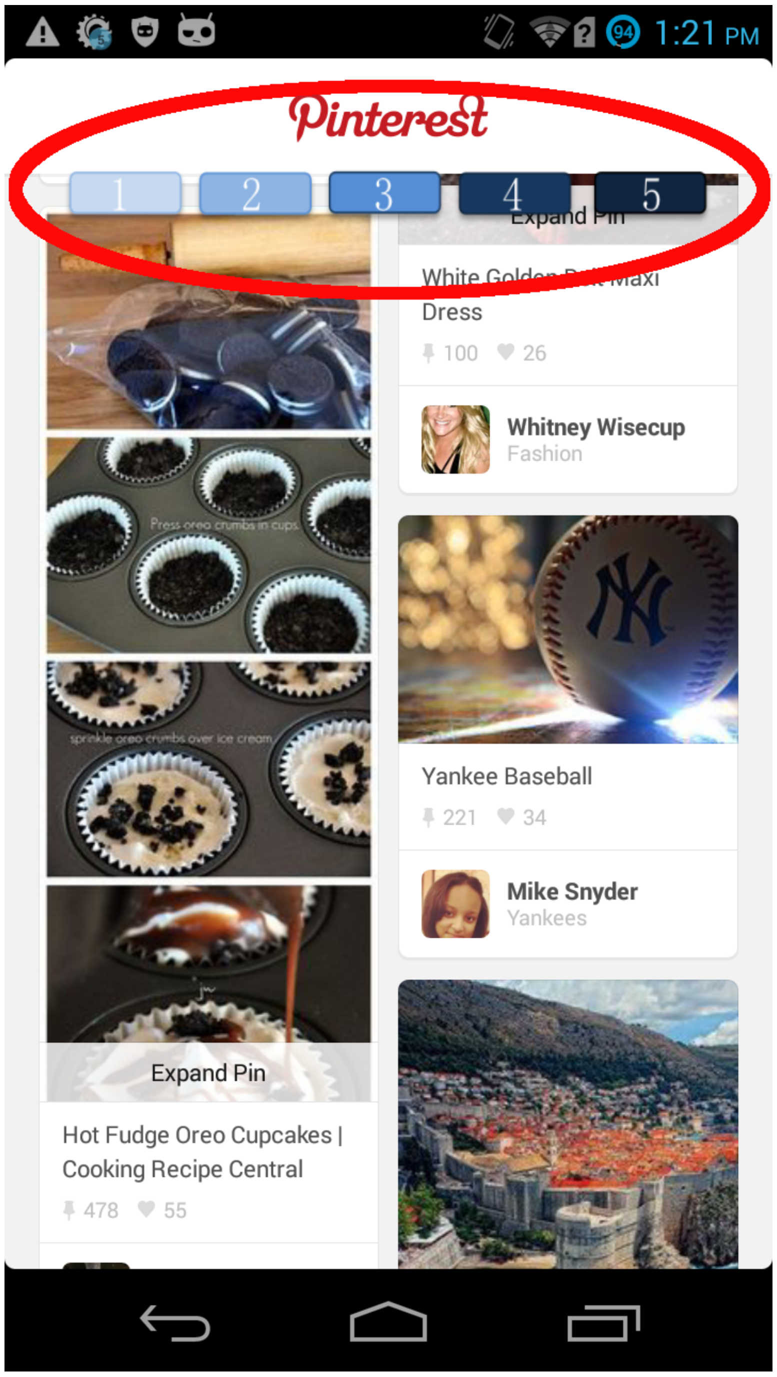


Figure 3.2: Rating overlay in its active state, as hovering over Pinterest.

App	Req. Rate [req/s]	Computation
MapQuest	0.89	Low
Pandora	0.38	Low
Pinterest	1.63	Medium
WeatherBug	1.09	Low
Google Translate	0.46	High

Figure 3.3: Applications in our study.

rating was prompted, current test case, arrival time and duration of the request, running application, OS metrics (such as CPU load), and network metrics (WiFi or cellular, packet drops).

3.2 User study

The goal of our user study is to understand how changing performance via delaying the network requests flowing from the frontend to the backend affects user satisfaction. Simply put, how much delay can we introduce before ordinary users employing popular applications become dissatisfied? To answer this and related questions, we leveraged the framework of Section 3.1 to augment popular applications. We then designed a study in which existing users of these applications could participate on their own phones in their normal environments.

Applications

We chose five of the most popular applications from the Google Play Store, applications where we believed we would have no trouble recruiting existing users. We also wanted to test applications that had varying request rates as well as varying amounts of "computation" that would likely be done on the datacenter. For the study, we chose MapQuest, Pandora, Pinterest, WeatherBug, and Google Translate as a representative sample of popular applications.

Subjects

Our study was IRB approved[1], which allowed us to recruit participants from the broad Northwestern University community. We advertised our study by poster at several locations on campus, and also advertised by email. Our selection criteria was that the subject had to own and regularly use a mobile phone capable of running our augmented applications, and the subject had to self-identify as a regular user of at least one of our applications. We selected the first 30 respondents who met these criteria for our study. As part of the study, each subject also filled out a questionnaire about their familiarity with phones, applications, etc. Each was given a $40 gift certificate at the end of the study.

Our 30 subjects have the following demographics, and the phones that they used are enumerated in Figure 3.4.

- 12 females, 18 males.
- 25 were age 17–25, 5 were age 25–35.
- 16 were in the engineering school, 10 were in the liberal arts school, 3 were in the journalism school, and one was in the education school.
- 25 had used a modern smartphone for at least 6 months, with 21 of these having used one for 2 or more years.
- MapQuest: 7 indicated a familiarity of 7 or greater (on a scale from 1 to 10).

[1]Northwestern IRB Project Number STU00093881.

- Pandora: 23 indicated a familiarity of 7 or greater.
- Pinterest: 11 indicated a familiarity of 7 or greater.
- WeatherBug: 8 indicated a familiarity of 7 or greater.
- Google Translate: 17 indicated a familiarity of 7 or greater.

Test Cases

We designed test cases, randomly selected periods of randomly selected additional delay, with an eye to inducing perceptibly different levels of performance in our applications as we ourselves perceived them. In a test case, each network request that occurs during a test case is delayed by a fixed amount. Our test cases all had a duration of one minute, and their delays were 0, 50, 250, 500, and 750 ms. Users were prompted for feedback in the middle of the test case (30 seconds in). Test cases themselves arrived randomly, with a user prompted an average of 152 times over the course of the study (one week). About 20% of prompts resulted in a response.

The only indication the subject had that a test case was running was being prompted, but the subject was also so prompted during the zero delay test case.

Methodology

The subject used his or her own personal smartphone for the duration of the study, albeit with our augmented test applications replacing the original applications. All logs were kept on the subject's smartphone, and were removed at the conclusion of the study, along with the augmented applications. The duration of the study was one week.

Model	Android version
Moto X	4.4.2
LG D500	4.1.2
Samsung Galaxy S3	4.3
Motorola Photon	4.0.2
Nexus 5	4.4.2
Samsung Galaxy S2	2.4.6
Samsung Galaxy S4	4.4.2
HTC One M7	4.4.2
Samsung Galaxy S4	4.4.2
Nexus 5	4.4.3
Samsung Galaxy S4 Mini	4.2.2
Samsung Galaxy S4	4.2.1
Samsung Galaxy S4	4.2.1
Samsung Galaxy S3 Mini	4.1.1
Samsung Galaxy S4	4.2.1
Nexus 4	4.2
Samsung Galaxy S2	2.3.6
Samsung Galaxy S3	4.3
Sony Xperia MT27i	4.0.4
Samsung Galaxy S4	4.3
Samsung Galaxy S5	4.4.2
Samsung Galaxy S4	4.2
Motorola G	4.4
Samsung Galaxy S3	4.3
Samsung Galaxy Note 2	4.2.2
Samsung Galaxy S3	4.3
Samsung Galaxy S3	4.3
HTC Droid DNA	4.4.2
Galaxy Nexus	4.2.2
Nexus 5	4.4.3

Figure 3.4: Model and Android version of participant mobile devices. With the exception of the 2 Samsung Galaxy S2 phones, all of the devices were considered modern phones at the time of the study.

When a subject first arrived, we had them fill out a questionnaire designed to determine their level of knowledge and comfort with a modern mobile device, as well as collecting demographic information. During this time we installed the augmented applications.

The subject was then instructed how to use the user interface for the duration of the study. This was done with a written document that was identical for all subjects. The document stressed that our interest was in the level of performance of the applications and not in their content. It did *not* indicate to the subject how performance might change. We indicated that it was important to provide feedback *about performance* whenever the interface flashed, and we described the intent of the scale as "1 being completely dissatisfied, 5 being the most satisfied, and 3 being neutral [with/about performance]". The subject would then leave the lab, and use their phone as they normally would for one week, answering rating prompts when appropriate.

At the conclusion of the week, the subject returned to the lab, and filled out an exit questionnaire. As they filled this out, we connected to their smartphone, downloaded the study data, and removed the test applications from their phone, replacing them with the original applications. Other than our interaction with them, and the user interface, changes to their normal experience of the applications was intentionally kept to a minimum.

3.3 Study results

Our study produced ratings from 27, and network traces from 29 of the 30 subjects. Recall from Section 3.1 that ratings could be provided as a result of a testcase prompt or inde-

Delay [ms]	Average Satisfaction	Average Change in Satisfaction	p-value for Comparison to No Delay
No Delay	4.0773	0	n/a
50	4.1000	0.0227	**0.002**
250	4.1233	0.0460	**0.001**
500	3.9408	0.1725	**0.022**
750	4.1975	0.1202	**0.005**

Figure 3.5: User satisfaction is largely unaffected by the introduction of delays of up to 750 ms into network requests made from Pandora, Pinterest, WeatherBug, and Google Translate. The p-values indicate that there was no statistically significant change in user satisfaction as request delay was added. The threshold here is 0.5 (10% of the rating scale).

pendently. In the following, we consider only the prompted responses, which correspond to test cases of any delay, including 0, on a scale of 1 to 5.

Given these constraints, our study produced 850 data points, each of which is the outcome of an intervention (the application of a test case) that resulted in a prompted response from the user. Given this number, as we decompose the results, for example by application or user, it is important that we highlight which conclusions have strong statistical support from the data. Hence, when we present p-values, we bold those that have $p < 0.05$ (95% confidence level).

To account for any anchoring effect due to user-based interpretation of satisfaction, we did our analysis based on the differences in satisfaction between delayed and undelayed application ratings for each user individually. In this way the comparisons that we make should be immune to differences in how users define their satisfaction levels.

Users tolerate significant added delay

If we look across all of our users and applications, we see the results of Figure 3.5. Here we record the average satisfaction for each level of delay, the average change in satis-

App	50ms	250ms	500ms	750ms
MapQuest	0.591	0.731	n/a	0.268
Pandora	0.133	0.127	**0.034**	0.291
Pinterest	0.131	**0.025**	0.101	0.356
WeatherBug	0.303	0.254	0.158	0.289
Translate	0.576	0.217	0.646	**0.000**

Figure 3.6: TOST apps, threshold = 0.5, p-values for testing that average satisfaction is no different for the given delay value and a delay of zero. Bold values are < 0.05.

App	50ms	250ms	500ms	750ms
MapQuest	0.488	0.416	n/a	0.127
Pandora	**0.000**	**0.004**	**0.000**	**0.002**
Pinterest	**0.011**	**0.000**	**0.003**	**0.034**
WeatherBug	0.105	0.071	**0.023**	0.055
Translate	0.155	**0.003**	0.292	**0.000**

Figure 3.7: TOST apps, threshold = 1.0, p-values for testing that average satisfaction is no different for the given delay value and a delay of zero. Bold values are < 0.05.

faction compared to that of the zero delay level, and a p-value. In aggregate, the data points to the possibility of introducing up to 750 ms of additional delay without having a significant effect on user satisfaction. The p-values reported are for a two one-sided t-test (TOST) [109], which measures how easily we can discard the null hypothesis that the mean satisfaction at a given delay level is *different* from the mean satisfaction at a delay of zero. In all cases, we can discard this hypothesis with at least 97% confidence. In such comparisons, the threshold of difference is also important to consider. The results in the figure are for a threshold of 0.5, or 10% of the 1–5 Likert rating scale we use. Given no other information, it appears very clear that we can add up to 750 ms of delay without changing the rating by more than 10%.

App	0ms	50ms	250ms	500ms	750ms
MapQuest	0.91	2.25	1.39	0.00	0.25
Pandora	1.10	0.96	1.22	1.08	0.88
Pinterest	1.08	1.49	0.95	0.89	2.06
WeatherBug	0.88	1.96	1.65	1.41	1.64
Translate	0.14	1.69	0.54	1.86	0.07

Figure 3.8: User satisfaction varies considerably across users. Rating variance across users for each combination of application and delay.

Delay tolerance for additional delay varies by application

We also considered the effects of introducing delay into individual applications, while still grouping all users together. Once again, we used TOST tests to identify where user satisfaction changed significantly compared to the no-delay case. These results are shown in Figures 3.6 (threshold of 0.5) and 3.7 (threshold of 1.0).

As one might expect, some applications experience more detrimental effects from introduced delays than others. For Pandora and Pinterest, we find that for a threshold of 1.0 (that is, one satisfaction rating) there is no statistically significant change in satisfaction caused by injecting delays ($p < 0.05$). For Google Translate more variation occurs, and for WeatherBug and MapQuest we can see that these applications are much more sensitive to additional delays. If we lower the TOST threshold to 0.5, we have less confidence that there is no change to satisfaction, although this may be simply due to the relatively small amount of data we were able to collect at this granularity.

Delay tolerance for additional delay varies across users

Figure 3.8 shows variance of user-perceived satisfaction for each of the delay levels. Recall that our test cases are randomly chosen and arrive at random times. What we are resolving here is that a given level of delay is likely to affect different users differently,

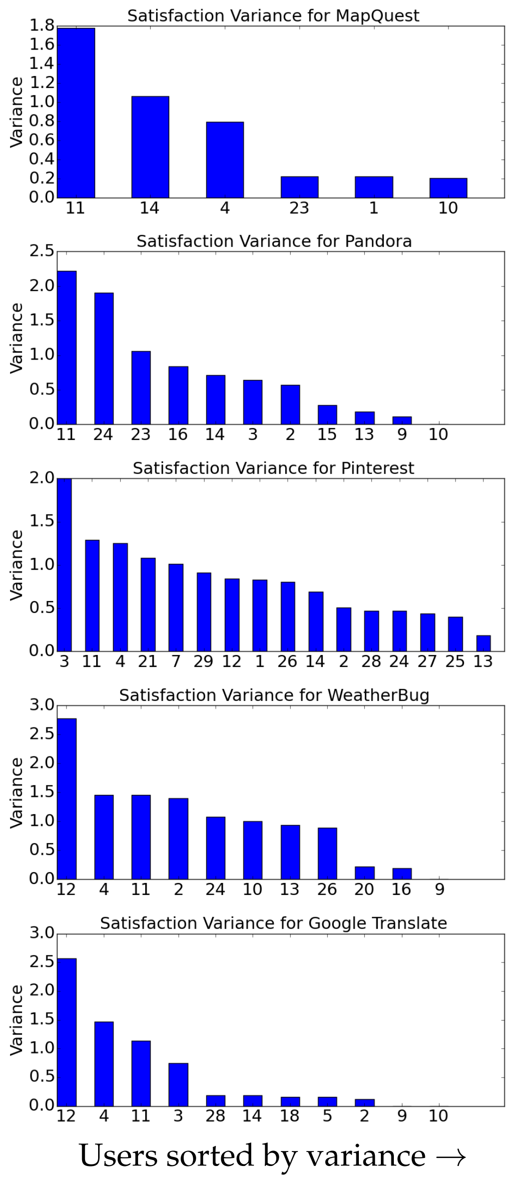


Figure 3.9: User satisfaction varies considerably across users. Variance of satisfaction for each user (horizontal axis), ranked by variance.

and also the same user differently across time.

Figure 3.9 illustrates this further. Here, for each application, we computed the variance of satisfaction for each individual user, aggregating over the different delays (which have equal probability). We then present each user's satisfaction variance, sorted by variance. We see that, for example, user 11 has the highest variance for MapQuest and Pandora, but is second or third in the rankings for the other applications.

Since the tolerance for additional delay varies across applications, and users, it seems natural that a real system should try to identify the more delay-tolerant users as particular opportunities for improving the request process.

3.4 Conclusions

We have considered the prospects for shaping the interactions between the frontends and cloud/datacenter backends of mobile applications. In particular, we considered delaying requests produced by the frontend and sent to the backend. Introducing such delays could be the mechanism for shaping the arrival process of the requests at the backend. Of course, too many delays or delays that are too large could irritate users.

There seems to be considerable opportunity to introduce delays without affecting user satisfaction. We developed a system that augments Android mobile applications with a delay component, and applied it to a range of popular applications. We then conducted an "in the wild" user study in which users employed our augmented applications instead of the ones they would normally use. The augmented applications would randomly add delays and accept user feedback about satisfaction. Analysis of the study data shows, among other things, the surprising result that delays of up to 750 ms can be introduced most of the time for most users without a major change in their measured satisfaction.

The results of this work may at first seem contrary to observations by Google [27, 53], Amazon [119], and others that suggest that even small increases in delay negatively impacts users who depart from the service. However, my work differs significantly from these works in at least two ways. First, we are considering mobile applications, not web applications on desktop environments. Much of the user interface of a mobile application quite smooth under delay as it is implemented on the mobile device itself, not on the backend. Second, we are soliciting the satisfaction of the user directly by prompting them, instead of indirectly by seeing if they stop using the application. We claim that users should be treated differently based on their *individual* tolerance for delay.

Chapter 4

To what extent can we shape traffic while not irritating users?

In this chapter I discuss the work that I did in developing and testing an algorithm that, by selectively introducing delays, can shape the characteristics of a user network request flow to be more like those of a desired distribution. In the previous chapter we found that delays of up to 750ms are tolerated by users, and so we use this as an upper limit of what acceptable delay.

We consider an approach which selectively delays requests originating at users to the datacenters backing their applications so as to shape the arrival process at the datacenter as Poisson arrivals (exponentially distributed interarrival times). This is a well-known arrival process particularly suitable for leveraging classic queuing analysis in the design of scheduling systems. We simulate this using the traces acquired from the user study. These traces also allow us to determine the likely effect on per-user satisfaction of our introduced delays. From this, we can evaluate the trade-off between our backend-centric goals for introducing the delays (Poisson arrivals), and our frontend-centric goals (maintaining user satisfaction with performance). There exist trade-off points where we can make the arrival process considerably more Poisson-like while not introducing delay that leads to dissatisfaction.

The contributions of this work are:

- We describe a potential algorithm that uses this headroom and varying tolerance to introduce delays that shape the user request stream.
- We evaluate this algorithm, in trace-based simulation, and find that there exist trade-off points where we are able to more closely match the Poisson arrival and rate limiting goals, while not reducing user satisfaction in a significant way.

4.1 System

We consider shaping user traffic while staying within acceptable boundaries. Given the results of the study of the previous Chapter, we selected 750ms as the amount of added delay that a user is willing to tolerate for most applications. We implemented a queue-based simulator which takes in user request arrivals, and delays each incoming request according to the specified shaping method. In this way, we are able to simulate shaping that might occur at any point along the request path, and stay agnostic to any final system, but rather explore the potential of shaping.

Figure 4.1 depicts how we envision the shaping to take place in a real world system. End users generate request workloads through their normal interactions with their applications—many actions on the part of the user will require additional content to be fetched from a datacenter, whether it be in the form of directly requested content such as text or images, or content that needs to be computed, such as travel routes, translations,

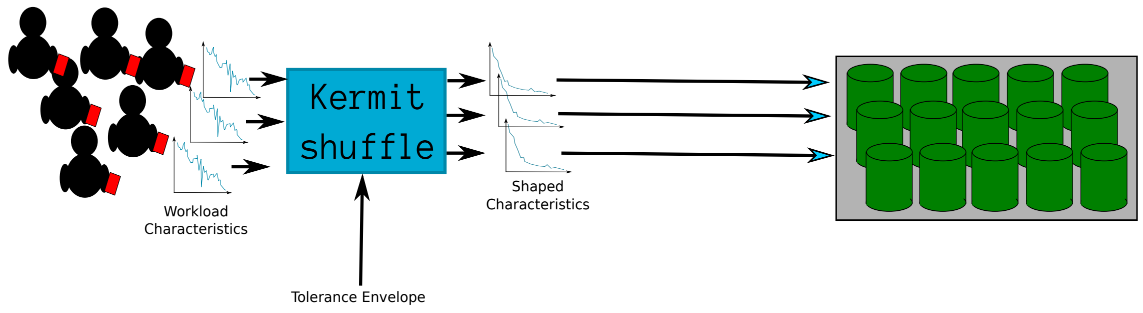


Figure 4.1: The kermit shuffle as we envision it in a real world scenario

etc. These request streams would pass through the Kermit shuffle, which is informed both by the value of the acceptable delay envelope of the user, and by the desired workload characteristics of the backend datacenter. Ideally, the request stream that would then flow out of the shuffle would have characteristics that were more in line with the desired ones, while staying within the users' envelope.

4.2 Algorithm

Our algorithm attempts to shape request arrivals such that they appear to have been drawn from a Poisson distribution. Poisson arrivals are a desirable property both in terms of easing the analysis of a system from a queuing theory perspective, and because they are less prone to the Noah Effect [136] in which traffic bursts aggregate across multiple timescales. It is important to note that other forms of shaping are certainly possible—our algorithm is intended as a demonstration of the concept of using the leeway provided by user delay tolerance to do shaping of some form.

The algorithm itself is based on a fairly simple principle: by capturing network request events, and delaying those events by amounts of time which are drawn from an exponential distribution, the new output departure times should leave with a pattern whose distribution is more akin to that of an exponential process. Put another way, if we consider the inter-arrival times of a user trace, and convolve those with the inter-arrival times of a known exponential process, the resulting inter-arrivals should look more exponential. The Kermit Shuffle algorithm takes as inputs an arrival stream, and a desired output interarrival rate, and is made up of three primary components, which loop the information through the system:

Input arrival estimator : As input events come in to the Shuffle, we must be able to estimate the current rate. We do this via an exponentially weighted moving average (EWMA), defined as

$$E_i = \alpha \times E_{i-1} + (1 - \alpha) \times (A_i - A_{i-1})$$

where E is the estimated rate, and A are the times of the arrivals, and α is in the range $(0, 1]$. The higher that α is set, the longer it takes for a change in rate to be reflected by the estimator, but by the same token this also means that outliers are less likely to throw off the estimate. We use this estimate as a potential basis for the exponential distribution from which we draw our delays.[1]

Output departure estimator : As requests leave the system, we track their rate with an EWMA. We then take the difference of this output rate and the desired output rate, and consider that the instantaneous error in our process, which we then use to inform our delay generation.

[1]The EWMA would "reset" if the interarrival time between requests became too long, as this indicated user inactivity. We deem active times "sessions" and explain them further in the text.

Delay generator : The meat of the Shuffle is the process by which delays are generated. From the two rate estimators, we know the rate at which inputs are arriving, and we know by how much our output rate is off. We start with the minimum of the input rate and the desired rate[2] and then apply what we call a "nudging factor" to it. The idea here, is that if the output rate is consistently too far off from the desired rate, we can try to increase the amount by which we are delaying. This nudging factor is calculated via a proportional-integral (PI) controller, which is defined as:

$$PI(t) = K_p \times err(t) + K_i \times \int_0^t err(\tau)d\tau$$

where K_p and K_i are the proportional and integral coefficients, respectively, and $err(t)$ is the error function at time t. The error is simply the difference between the output rate and the desired rate. We add the output of the PI controller to the previously determined rate, while ensuring that the rate stays non-zero. This nudged rate is then used to generate a delay from an exponential process with that rate, and added to the arrival, which emitted. The entire process is shown algorithmically in Figure 4.2, and schematically in Figure 4.3.

4.3 Evaluation

We evaluate the algorithm in simulation by feeding it with traces that we collected during the user study. The traces collected contained tuples of the form

$$\{time, appName, user, totalTime, delay, testCase\}$$

[2]The minimum, because if the input rate is lower than the desired rate, we cannot speed things up!

```
1:  procedure SHUFFLE($arrival, inter_d$)
2:      $inter_u \leftarrow$ EWMA(input)          ▷ Estimate interarrivals of input using EWMA
3:      $inter_s \leftarrow$ EWMA(output)         ▷ Estimate interarrivals of output using EWMA

4:      $m \leftarrow min(inter_u, inter_d)$      ▷ Limit the shaping interarrival
5:      $err \leftarrow inter_s - inter_d$        ▷ how much is the shape off
6:      $nudge \leftarrow PI(err)$                ▷ nudge with PI controller
7:      $m \leftarrow max(0.001, m + nudge)$      ▷ ensure nudging is not negative
8:      $delay \leftarrow exp(m)$
9:      $arrival$ += $delay$
10: end procedure
```

Figure 4.2: Kermit Shuffle User Traffic Shaping Algorithm.

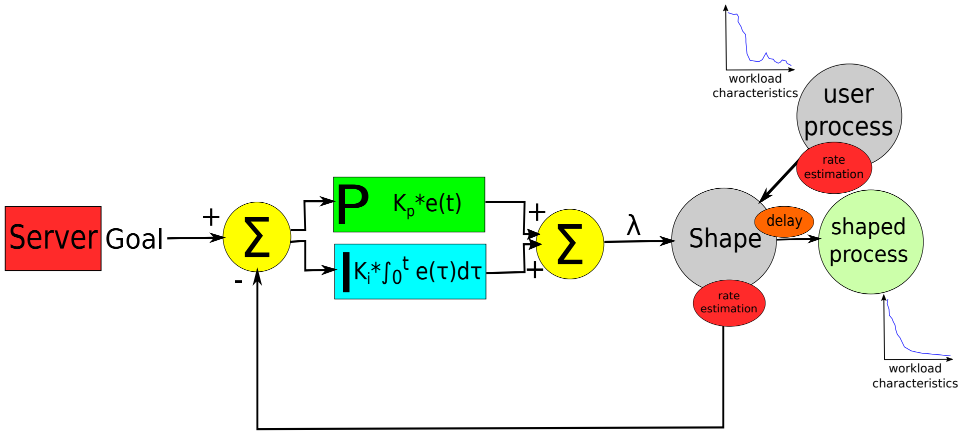


Figure 4.3: Schematic view of the Kermit Shuffle algorithm, which makes the estimation and feedback of data clear.

Recall that each test case ran for one minute, and during this time *all* network requests were delayed. For this reason, we have considerably more requests than test cases, and for our simulation we use all 140,401 collected records. For each simulation, the trace was segmented into "sessions", where a session was defined to be any section of requests where no interarrival was greater than 60 seconds. In other words, we consider bursts of arrivals that correspond to application activity that is typically driven by the user.

For each session, we compare the original and shaped session using quantile-quantile (or Q-Q) plots, an example of which is shown in Figure 4.4. The desired shape in each plot is the solid diagonal line. If the points were to line up exactly on that line, we could confidently say that the two data sets plotted were drawn from the same distribution. We fit a line to each graph using a least squares fit and quantify the fit with an R^2 value. We can then compare the original and shaped graphs via the difference in R^2 values, which we refer to as ΔR^2. This delta is defined as

$$\Delta R^2 = R^2_{shaped} - R^2_{input}$$

The closer R^2 is to 1, the closer the quartiles of the trace are to being on the desired diagonal in the Q-Q plot, so if the change in R^2 is positive, this means that we have made the interarrivals more exponential-like, and the bigger the change, the more effect we have had. In the figure, the ΔR^2 is 0.1822. Remember that R^2 ranges from 0 to 1, so this represents a fairly significant shape in changing.

It is important to note that evaluations were done across sessions, in order to capture local shaping (as sessions may have differing rates), however the algorithm has no awareness of such structure, and simply operates on the rate provided via the estimators, thus

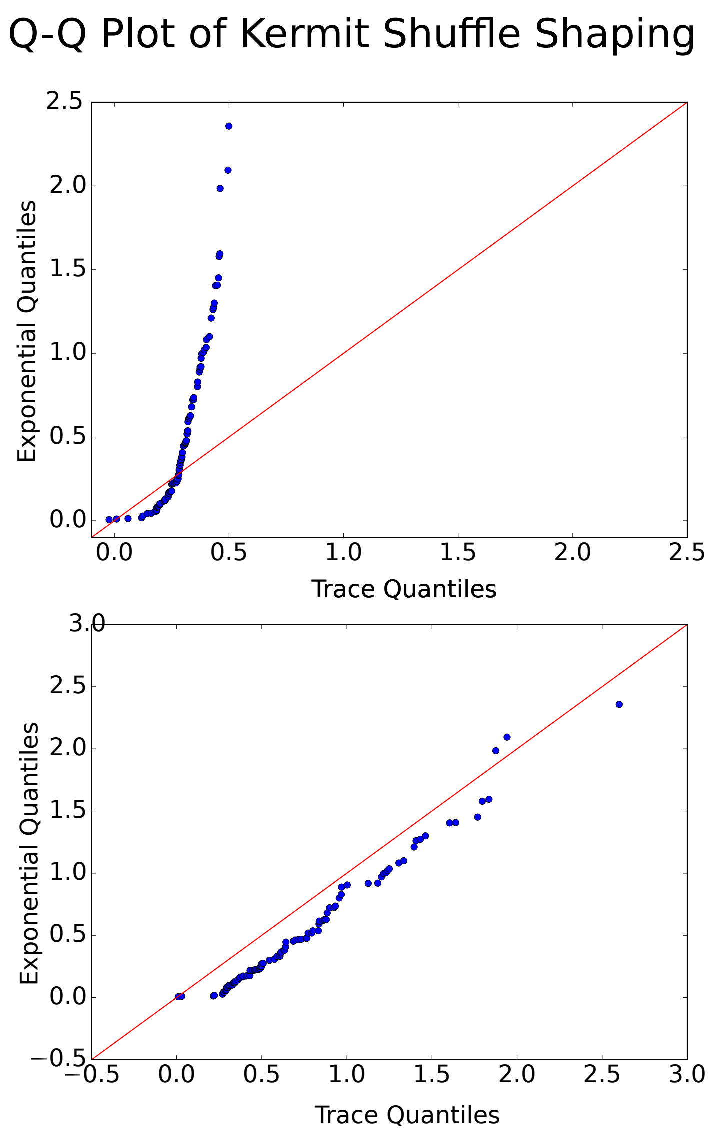


Figure 4.4: Example of the effect of the algorithm on shaping an input. The input (top graph) has been improved by$\Delta R^2 = 0.1822$, leading to the much more Poisson-like output (bottom graph).

needing no *a priori* knowledge.

For each of the evaluations, the shaping was run 30 times on each trace, and the results averaged in order to estimate the ensemble average behavior. In addition, for each session the Q-Q comparison is run 30 times, as the distribution being compared to in the plot is itself randomly generated each time. For the PI controller and EWMA estimator, we conducted multiple runs and settled on constant values of $K_p = 0.9$, $K_i = 0.5$, $\alpha = 0.8$ as optimal.

We additionally want to see how the algorithm performs across varying loads. To do so, we set the desired rate according to a variable system load, where load is defined as $load_{sys} = \frac{inter_{desired}}{inter_{trace}}$, and $inter_{trace}$ was defined as the average of the trace. It is important to note that this information is not needed during shaping, it simply provides us with a method of evaluating performance across various loads.

Individual users can be shaped mildly

Figure 4.5 presents the results of running the algorithm on an individual user. For each load factor, we report the average ΔR^2, the average delay introduced to each request, as well as the 95^{th} and 90^{th} percentile of the delay introduced. For this evaluation, we consider the point where the 95^{th} percentile delay grows beyond the acceptable tolerance envelope as the limit for shaping.

As a reminder to the reader, we simulate differing load conditions by setting the desired output interarrival relative to the incoming interarrivals in order to test how the Kermit Shuffle performs at differing loads. We can see that as the load level increases towards 1.0, the ability of the shuffle to shape towards exponential interarrivals increases. This trend is encouraging, as the ability to shape traffic becomes more meaningful as load levels increase. Consider a canonical datacenter—as the load of incoming traffic approaches 1.0, it is likely for queuing delays to start accumulating, and if this load stays above 1.0 for any meaningful amount of time, the delays will begin to increase dramatically.

We found that the average R^2 for the sessions of the user trace was around 0.58, so we would not expect to see ΔR^2 of greater than 0.42 in the optimal case. We can see from the figure that for User 1, the limit occurs at a load of 0.5, with $\Delta R^2 = 0.0665$. If we consider keeping the *average* injected delay below the envelope, we can shape the trace with $\Delta R^2 = 0.1204$. Given our bounds, this is relatively good, but ultimately we would like to be able to do better.

We ran the same analysis on each user from the study, and the results are enumerated in Figure 4.7. For all user traces, the R^2 of the case was generally between 0.5 and 0.6. We can see that the ability to shape and the load at which we can shape traffic without annoyance varies quite a bit between users, which indicates that shaping at the user level will produce the most beneficial results overall. As in the case of the single user, the ability of the Shuffle to shape increased with rising load, but the actual load at which tolerable delays occur varies.

Aggregated users can be shaped minimally

Because the matter of where shaping would be most optimal is still an open question, we also wanted to test how the Kermit shuffle would perform when we aggregated the user traffic traces together and pushed them all through the Shuffle, as a simulation of the Shuffle dealing with traffic coming in from multiple users.

The results of running the Shuffle on all the traces are shown in Figure 4.6. Much the same as with individual traces, ability to shape increases with load, however the ΔR^2 ends up being significantly lower, roughly half of that of the previous results. We see that the limit of shaping for aggregate users occurs at a lower load of 0.1, with a ΔR^2 of 0.0053,

Load	ΔR^2	Avg Delay	95 %ile	90 %ile
0.1	0.0102	0.0389	0.1479	0.1054
0.2	0.0236	0.1014	0.2963	0.2118
0.3	0.0408	0.0947	0.4448	0.3186
0.4	0.0537	0.2548	0.5925	0.4239
0.5	**0.0665**	**0.2206**	**0.7403**	**0.5324**
0.6	0.0771	0.2458	0.8949	0.6394
0.7	0.0892	0.4656	1.0505	0.7472
0.8	0.0990	0.2271	1.1949	0.8515
0.9	0.1071	0.3547	1.3407	0.9604
1.0	*0.1204*	*0.4582*	*1.4912*	*1.0669*

Figure 4.5: Shaping of trace of User 1. Ability of the algorithm to shape an individual user's traffic increases with load. For this user, the algorithm can produce a $\Delta R^2 = 0.0665$ while staying within the delay tolerance envelope supported by the user study 95% of the time (bold). It produces $\Delta R^2 = 0.1204$ by staying in envelope on average (italic).

Load	ΔR^2	Avg Delay	95 %ile	90 %ile
0.1	**0.0053**	**0.1316**	**0.4518**	**0.3246**
0.2	0.0101	0.2632	0.9048	0.6494
0.3	0.0170	0.3948	1.3543	0.9654
0.4	0.0223	0.5265	1.8060	1.2982
0.5	*0.0299*	*0.6581*	*2.2828*	*1.6272*
0.6	0.0337	0.7898	2.7273	1.9482
0.7	0.0386	0.9214	3.1769	2.2785
0.8	0.0462	1.0531	3.6138	2.6083
0.9	0.0509	1.1847	4.0911	2.9127
1.0	0.0552	1.3163	4.5210	3.2436

Figure 4.6: Ability of the algorithm to shape aggregate traffic increases with load. Bold indicates staying within delay tolerance 95% of the time. Italic indicates staying within delay tolerance on average.

and 95^{th} percentile delay of 0.4518. Given that the R^2 of the aggregate traffic is around 0.61, this represents a significant decrease in shaping ability, and potentially suggests that, at least for this algorithm, it may end up being more fruitful to shape at the individual user level.

User	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ΔR^2	0.067	0.012	0.007	0.017	0.000	0.012	0.019	0.005	0.008	0.005	0.011	0.012	0.006	0.005	0.008
Load	0.5	0.2	0.2	0.2	0.1	0.5	0.2	0.1	0.1	0.1	0.2	0.3	0.1	0.1	0.1
User	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
ΔR^2	0.006	0.018	0.012	0.003	0.000	0.056	0.009	0.009	0.009	0.033	0.003	0.039	0.009	0.024	n/a
Load	0.1	0.3	0.1	0.1	0.1	0.4	0.1	0.1	0.2	0.3	0.5	0.3	0.2	0.2	n/a

Figure 4.7: Shaping results for each individual user, showing the maximum load and ΔR^2 achievable while staying within the 750 ms delay tolerance envelope supported by the user study 95% of the time.

4.4 Limitations

If we take a step back and think about the meaning of our above findings, we notice that we can indeed produce results that stay inside of the apportioned user envelope, but perhaps not in the most critical of times. If we think about need for energy savings at a datacenter, our shaping would be most meaningful if it could produce effects at loads that approach 1, that is, when the datacenter is close to or becoming overloaded. The shaping power of our approach does scale as load goes up, however we are unable to stay within acceptable delays. Below I describe several attempts that we undertook to solve this problem.

Alpha shaping

When we began to dig into how the elements of the algorithm were behaving, we noticed that the output of the PI controller had a tendency to go "off the rails", and sink into extreme negatives for long periods of time. We noticed that these events typically coincided with times when the rate estimator would spike to large values. An example of this is shown in Figure 4.8.

Despite us setting the α value relatively high (0.8), our rate estimator was behaving rather erraticaly. Due to the nature of using a randomly generated value for the delay, our thought was that we were giving too much power to the instantaneous variation of

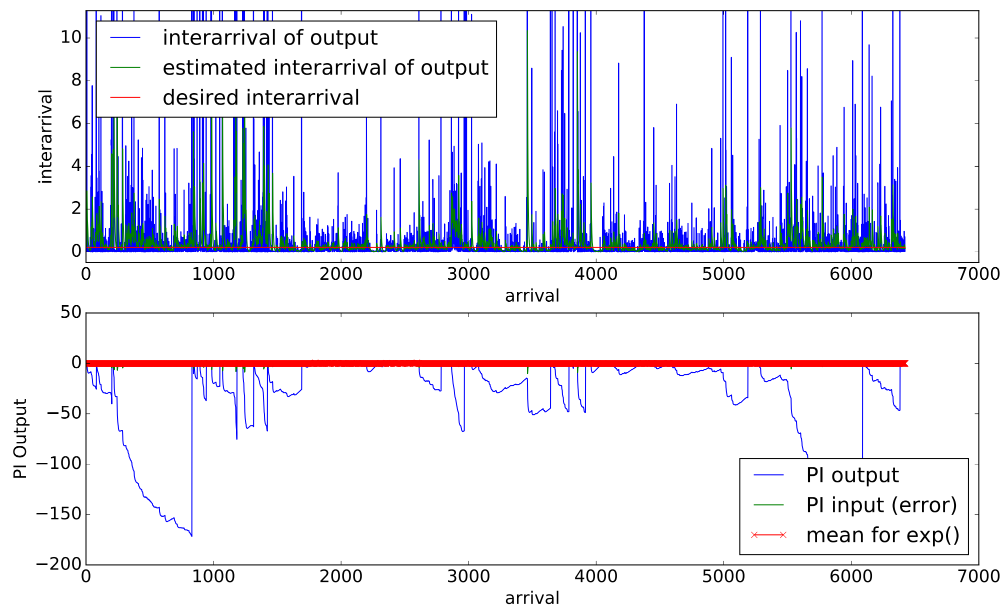


Figure 4.8: With a set α the output of the PI controller can go off the rails

added delays with respect to rate estimation, especially during bursts of activity. When our initial attempt of simply scaling up α to higher values like 0.95 did not correct this behavior, we began exploring dynamic methods of setting the α-factor. The shapes are shown in Figure 4.9 and results enumerated in the text below. The idea behind all of these shapes is to scale α down as m, the rate of the exponential, goes up.

Linear scaling : Our first idea was to simply linearly scale α down as the exponential rate increased. The results of this function are shown in Figure 4.10. While α does respond to the rate, it does not seem to be enough to overcome the volatility of the arrivals. The equation used to generate the figure is:

$$\alpha(m) = 0.95 - 4.0 * m^3$$

[3]we cap α to never go below 0.

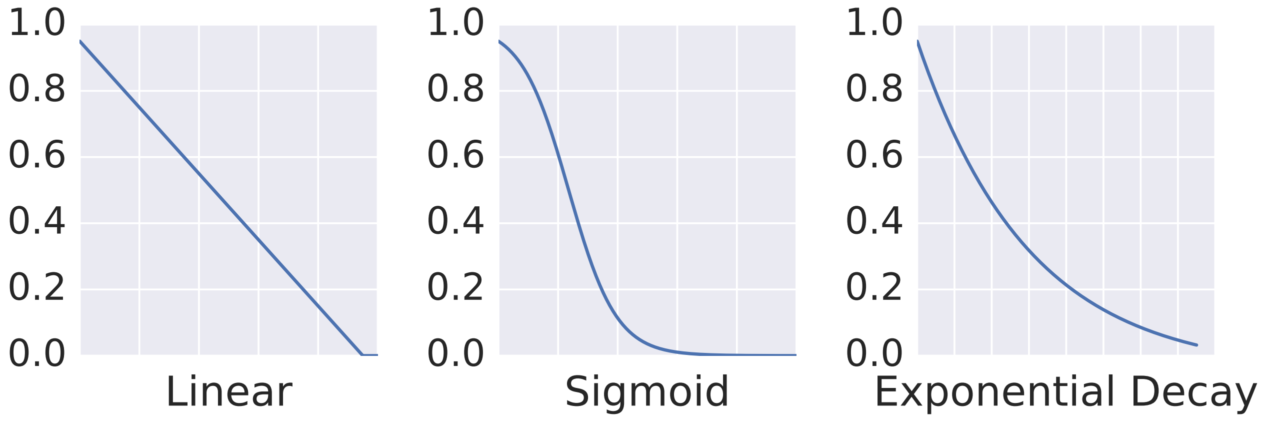


Figure 4.9: The three different functions used to try to scale down the α factor of the EWMA for higher exponential rates.

Shaped sigmoid function : The sigmoid function experiences a much steeper fall than a simple linear relationship, so this is what we tried next, the results are shown in Figure 4.11. Despite the scaling effects, we still seem to have large PI error buildup. The shaped sigmoid function used to generate the figure is:

$$\alpha(m) = \frac{1}{1 + e^{0.05* - m}}$$

Exponential decay : Our final attempt was to use an exponential decay function, due its much higher initial dropff, the results are presented in Figure 4.12. Once again, our scaling factor is unable to effect any reduction in the building of PI error. The function used to generate the figure is:

$$\alpha(m) = -(1 - 0.95) * e^{\frac{1}{0.05} * - m}$$

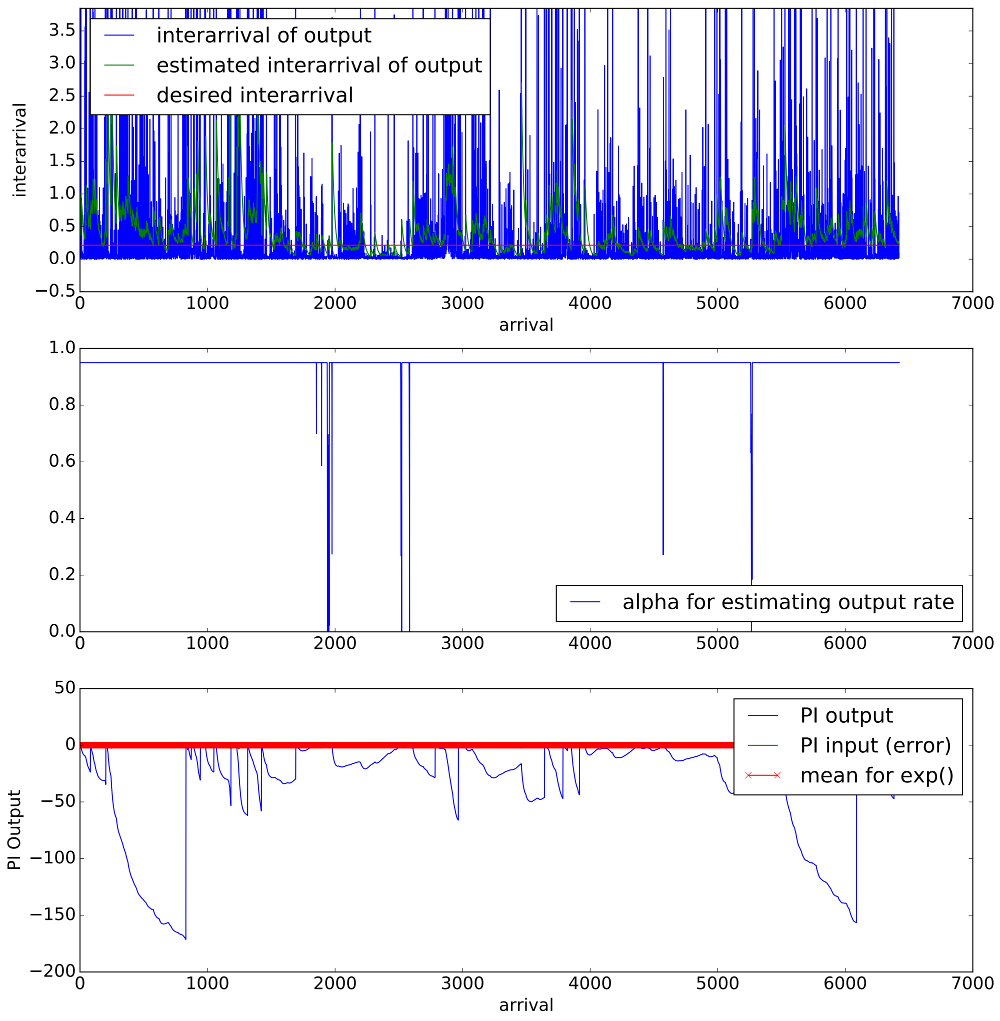


Figure 4.10: Linear α scaling is ultimately unable to reduce PI error accumulation

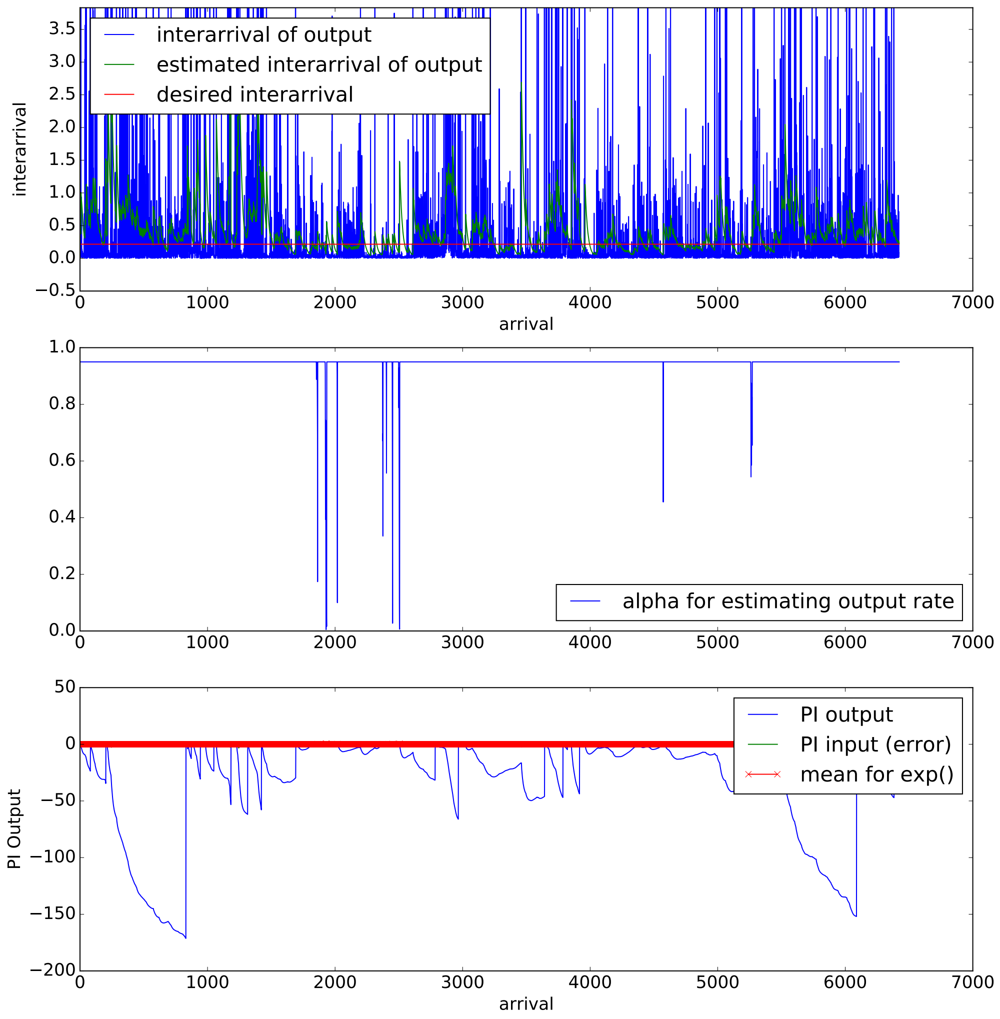


Figure 4.11: Shaped sigmoid α scaling is ultimately unable to reduce PI error accumulation

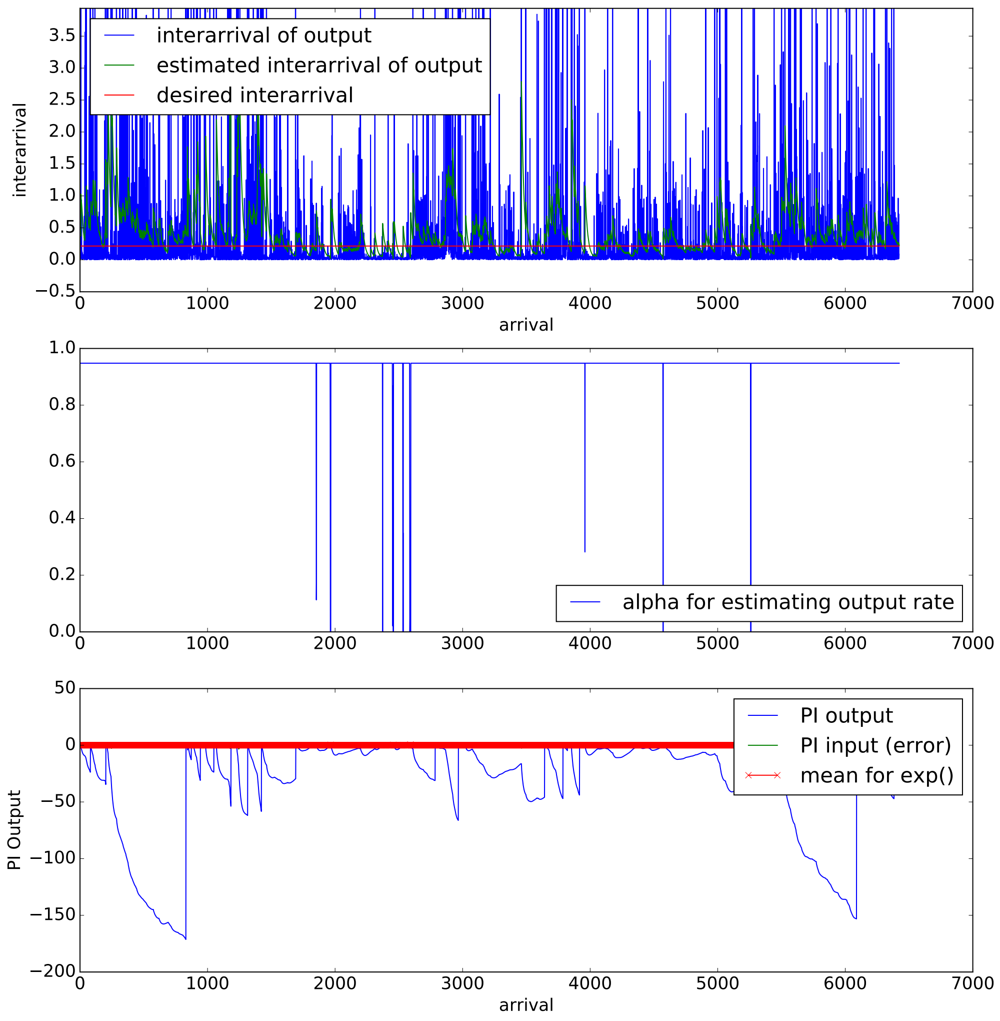


Figure 4.12: Exponential α scaling is ultimately unable to reduce PI error accumulation

Evaluation strategy

One additional thing we realized here is that our method of evaluating distribution fit with a linear regression / Q-Q plot approach may not be the best one. Since we are constantly varying the mean of the exponential used to generate the delays, we my be producing an output whose characteristics cannot be compared to a singly generated distribution. After all, what does it mean to be producing outputs whose departure times are drawn from distributions of dynamically varying means? We would like to further address and explore this as future work.

4.5 Conclusions

As a proof of concept of user traffic shaping, we developed an algorithm that introduces delay to requests in a controlled manner to attempt to make their arrival process at the backend have exponential interarrival times (Poisson arrivals). We simulated the algorithm using the trace data from the study. While keeping the introduced delays within the tolerance determined by the study, the algorithm is able to appreciably affect the arrival process, pushing it significantly closer to Poisson arrivals.

Chapter 5

Does environmental prompting change the delay tolerance envelope?

In this chapter, I describe work that was done in designing a user study which would both replicate the results of the previous study on a larger study population, while letting individual users control their own delay settings, and in testing the power of environmental prompting.

In the previous chapter, I described a user-study that was performed to explore the space of the user delay tolerance envelope, wherein we found that for a population of students at Northwestern University, delays of up to 750ms could be introduced to cloud-backed mobile applications with no statistically significant loss in user satisfaction. Given previous work on correlating delays with user satisfaction, this result was surprising, and merited further study. Given our collaboration with Peking University, we had ready access to a distinct study population in the pool of university students from there.

To the best of our abilities, and to the extent that the PKU user-study framework allowed, we recreated the user-study of the previous chapter, in order to strengthen our belief in the results of that study. We leverage the same toolchain to interpose on existing popular Android applications. In order to choose applications that would be popular with our study demographic, we selected popular applications from the Wandoujia [132] Android application market, such that their characteristics closely matched the applications from the previous study. Using the toolchain, we modify these popular applications such that interactions between the frontend and backend flow through code under our control, allowing us to inject delays into the applications. Our code also adds a method for users to control their delay setting, and users are asked to set their delay as high as possible while not annoying them. A core outcome of this study is that, even when users were allowed to set their own delay, we see a similar level of acceptable delay as the envelope, that is, values around 750ms.

Additionally, we wanted to test our theory that environmental prompting would increase the delay tolerance envelope. By "environmental prompting" we mean notifying the user that their actions had direct environmental repercussions—specifically telling them that the higher they set the delays, the more environmentally friendly they were being. Users from the study were randomly placed into reconsidered or considered groups, wherein the only difference was whether that group would be environmentally prompted or not. A fascinating result is that there was no statistically significant difference in delay settings between the two groups.

The contributions of this work are as follows:

- We show, via a user study involving 70 users running 5 popular Android applications in a controlled lab study, that even when users are asked to set their own delay, that the delay tolerance envelope is large. This envelope varies by user, and by application.

- We evaluate several methods for analytically determining any given users envelope.
- We show, using the same study, that there is no statistically significant difference in the envelopes of users who have been environmentally prompted vs. those that have not been.

5.1 Frontend augmentation

Our user study is based on popular Android Java applications that are available as Android application packages (APKs) from the Wandoujia mobile marketplace. The process and architecture of the application augmentation is based on the same process as that of Section 3.1, with a few notable differences. In addition to the delaying proxy, the framework introduces a separate component, DelayController, which displays and controls the current delay setting on the mobile phone. The DelayController functionally sits in the same place as the FBCollector, and the functional flow of the whole augmented application is very similar to that of Figure 3.1.

DelayController sits between the DelayController and Rate-Message Receiver, and provides the same flow of information through the augmented application as FBCollector did. Instead of providing a scale for rating satisfaction though, DelayController creates an display interface, which is shown in Figure 5.1. The interface is comprised of two main parts:

1. The taskbar notification icon. Shown in the left-hand side of the figure, the notification icon displays what percentage of delay is currently being introduced into the application.

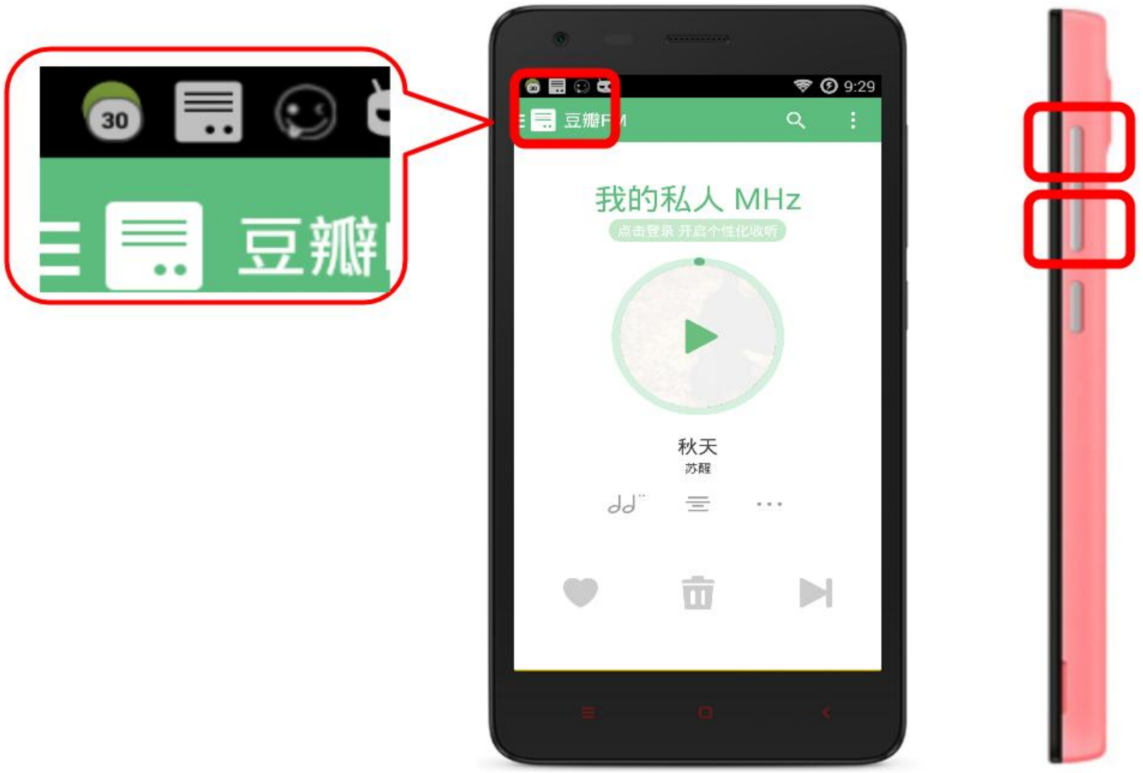


Figure 5.1: DelayController interface, which shows the user the current delay setting (30 %), and allows the user to control the delay using the volume keys, shown over Kingsoft.

2. The user-accessible controls. DelayController catches when the user presses the volume up / down keys, and uses those events to change the delay setting. The volume up key will increase the delay, and the volume down key will decrease the delay.

In addition to the interface on the mobile device, study subjects were assigned to a computer terminal in the study environment, where they would be given instructions via a web-based interface. This website would also be responsible to notifying the user of the beginning, end, and mid-task notifications.

5.2 User study

With this study we want to answer 3 fundamental questions:

1. Will we find the same delay tolerance envelope (750ms) of the study in Chapter 3 in a distinct user population?
2. Will we find a similar envelope, *even while giving users direct control over the delay?*
3. Will we find any differences between a prompted and an unprompted population?

In order to answer these questions, we used the interpolation framework of Section 5.1 to augment 5 popular applications, and designed a study in which existing users of those applications would be allowed to set their own delays, while accomplishing pre-set tasks in those applications.

Applications

We chose 5 of the most popular applications from the Wandoujia application marketplace, applications where we would have no problem finding existing users. We wanted to choose applications that were similar to the applications of the user study from Chapter 3, so that similarities in tolerances would be likely to be meaningful. We chose the following applications as our representative application set for this study

- Douban Music: A streaming music application connected to Douban, a social network. Similar to Pandora.
- RenRen: A social networking application where users can post status updates and pictures. Similar to FaceBook and Pinterest.

- KingSoft: An application in which users can input words or phrases to translate to and from various languages. Similar to Google Translate.
- Weather: An application for finding weather conditions and forecasts. Similar to WeatherBug.
- Youku: A service where users can upload and watch videos. Similar to YouTube[1]

Subjects

Peking University does not have a formalized IRB process, however we designed the study in a very similar way to the original US study, and made sure to take the same steps to preserve subject anonymity and not collect any personally identifying information. We advertised to the broader Peking University population via flyers posted in computer labs and University hallways / bulletin board, and email advertisements. Our selection criteria was that the subject had to be familiar with mobile phones, and have familiarity with at least some of the applications that they would use during the study. We selected the first 70 participants who responded and qualified. As part of the study, each subject would also fill out entrance and exit questionnaires, which included demographic information. At the end of the study, each participant was rewarded with a gift card worth 35 Chinese Yuan (approximately 5 USD). The demographics are enumerated in Figure 5.2.

[1]We attempted to augment a map application similar to MapQuest, but the two popular applications, BaiduMap and GaodeMap both used hash checking code in the application that prevented augmented versions from being created.

Users

Age	18-25	45
	25-35	25
Gender	Male	25
	Female	45
Area of Study	Computer Science	7
	Science / Engineering	20
	Medicine	9
	Law / Policy	16
	Other	18
Length of smartphone usage	0-1 Months	0
	1-6 Months	0
	6 Months - 1 Year	1
	1-2 Years	2
	2+ Years	67
Smartphone Type Owned	Android	39
	iOS	31
Carrier	China Mobile	45
	China Unicom	25

Figure 5.2: User study demographics.

Methodology

Each subject would use a lab phone provided to them, and complete all tasks in the lab. Logs would be kept on the phone as well as a coordinating server, and be tagged with an anonymized user identification number. The duration of the study was approximately one hour.

When the subject arrived, the proctor would take them to their station, and explain how the study was laid out. Each task would take 10 minutes, and was split into 3 phases:

- 0 - 2 min: This would be time when a subject would familiarize themselves with the application and task, in case they had not used this task before. During this time the subject was asked not to change the delay[2].

[2]Although the user was asked not to change the delay during the introductory 2 minutes, some users

- 2 - 6 min: At the 2 minute mark, the subject would be prompted via an alert on the website that they could now change the delay in the application, and would be asked to set the delay as high as possible, such that it did not annoy them or interfere in their ability to complete the current assigned task. Depending on which study population the subject had been assigned to, they would also be reminded that higher delay settings would translate to more environmentally friendly behavior.

- 6 - 10 min: At the 6 minute mark, the subject would be reminded via an alert on the website that they could use the phone interface to change the delay at any time. We were careful not to tell the subject to try to increase the delay at this point, as that might bias how high they set it. At the 10 minute mark, the subject would be told that the current task was now complete, and they would be prompted to continue on to the next page in the instructions.

Testbed

The user study was set up as a lab-controlled study—subjects would enter the lab, and a proctor would take the subject to their station, and explain to them how the study would take place. Each station was set up with the phone to be used: A Xiaomi Hongmi 2A phone with CyanogenMod 11 (a popular free Android ROM based on Android 4.4.4) installed on it. The station also had a desktop computer, which had a browser open to the study website. The study website would provide the user with instructions for the study, instructions for each tasks, and any necessary prompts, as well as relay information to

did change the settings here. We do not consider these levels in our analysis

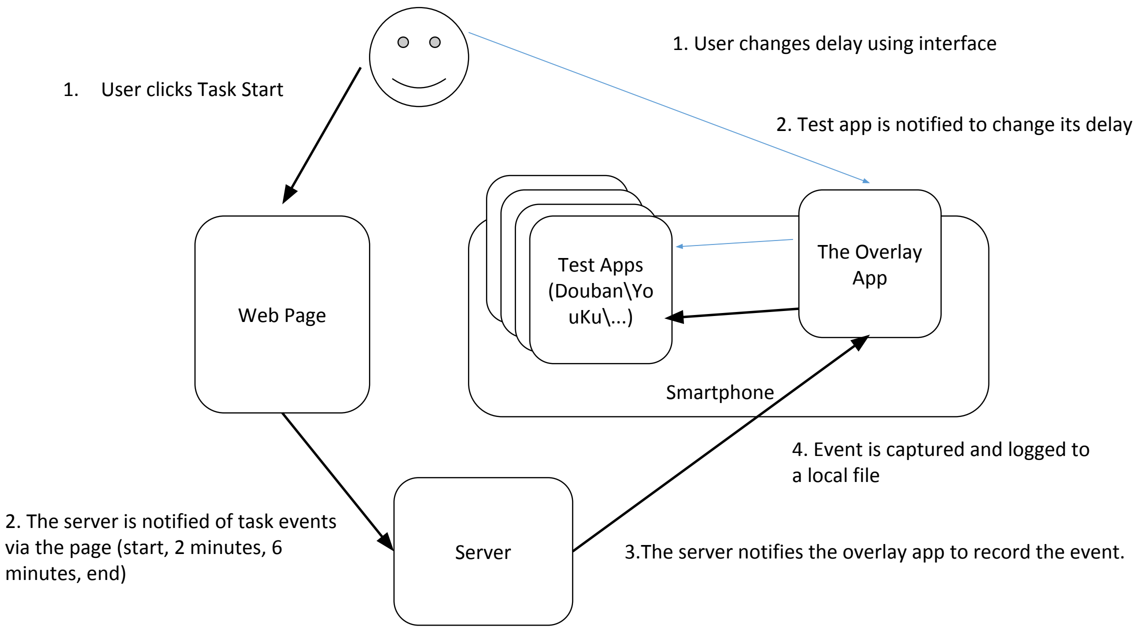

1. User clicks Task Start
1. User changes delay using interface
2. Test app is notified to change its delay
Web Page
Test Apps (Douban\YouKu\...)
The Overlay App
Smartphone
4. Event is captured and logged to a local file
2. The server is notified of task events via the page (start, 2 minutes, 6 minutes, end)
Server
3.The server notifies the overlay app to record the event.

Figure 5.3: Set up of testbed for the PKU study, showing the flow of information between the user, mobile device, instruction website, and coordinating server.

the coordinating server. All interactions between the user, phone, website, and server are displayed in Figure 5.3.

When a user began a task, the instruction website would create a timer to measure the progress in the current task. At the 0, 2, 6, and 10 minute marks, the site would do two things: 1. Send a message to the coordinating server, with a message as to which event had just occurred, and 2. prompt the user with an alert, with the message pertinent to the phase of the task they were at.

The coordinating server would log all events, in order to keep redundant information in case of any message loss, and would also send a message to the mobile device. The DelayController component of the application would receive this message, and log all received events in a file as a tuple of the form:

$$\{timestamp, appname, event\}$$

Whenever a subject changed the delay via the interface, the DelayController would also log the change via a tuple of the form

$$\{timestamp, delay\}$$

As well as logging each network request via a tuple of the form

$$\{timestamp, totalTime, extraDelay, delaySetting^{3}, screenOn\}$$

Once a subject finished all tasks and the exit questionnaire, all of the logs would be collected from the device, validated against the logs of the coordinating server, and packaged up for analysis.

5.3 Tasks

In order to simulate real-world conditions, we designed tasks for subjects to complete in each chosen application:

- **KingSoft**: The subject was given a passage from a freely available copy of "Alice's Adventures in Wonderland" by Lewis Carroll, and asked to translate the passage

[3] We measured the desired amount of delay, as well as the actual amount of delay that was injected, which varied slightly due to lack of guaranteed precision in the delay mechanisms available. For example, in one collected request, the delaySetting was 600.0, but the recorded extraDelay was 610.0

one sentence at a time.

- **RenRen**: The subject was asked to browse content from their friends in the application, and to browse through their various photo albums.
- **Douban Music**: The subject was asked to find and listen to various songs, switching to new songs after every 30 seconds or so.
- **Youku**: The subject was led to a channel that contained short videos (on the order of 30 seconds), and asked to watch those videos.
- **Weather**: The subject was given a list of locations, and asked to find both the current conditions and weather forecast of each of them.

5.4 Study results

Our study involved 70 subjects, and produced meaningful data from 67 of the subjects. For each subject, and each application, we collected the times of the events for each task (begin, 2 minute prompt, 6 minute prompt, end), the time and delay level of any time the user changed the delay, as well as properties about each network request that was sent by the study applications.

In total, the study produced 7780 delay setting changes, which we use as the basis for the analysis. One obvious question is—how do we determine what a user's ultimate tolerance envelope was from a time-series of setting changes? We explore a few different methods of determining the envelope. As we decompose the results, we bold findings which findings have strong statistical confidence behind them, that is, findings with p-values of lower than 0.05 (95% confidence level).

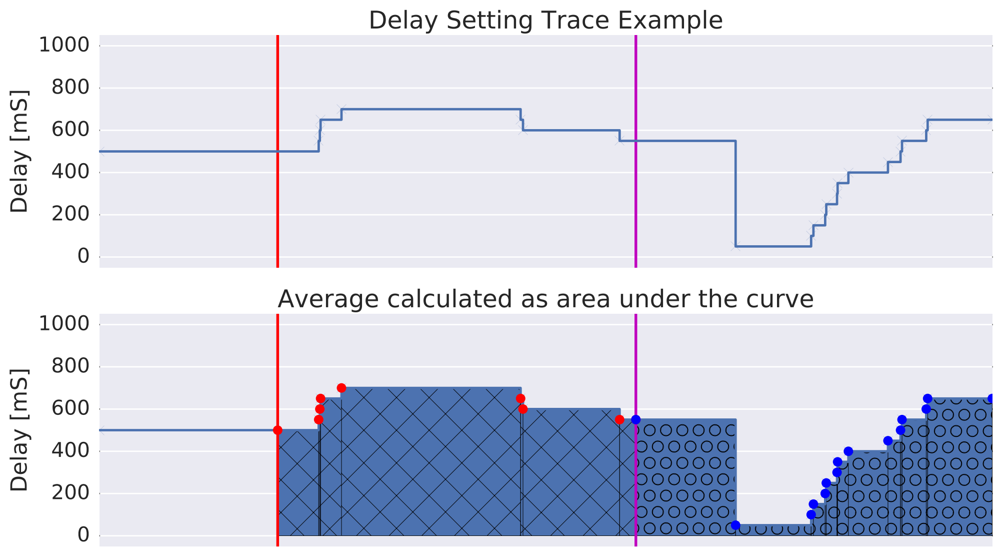


Figure 5.4: Calculating the delay tolerance envelope using the area under the curve of the delay setting over time. Left vertical line indicates 2 minute prompt, right vertical line indicates 6 minute prompt. Average for this curve is 524.64

Area under the curve

Our first method of evaluating the tolerance envelope is by treating the recorded delay settings during a task (the 10 minutes one user is using one application), and calculate the average using the area under the "curve". An example of this is shown in Figure 5.4. We calculate the average of 3 distinct time-periods:

- *Considered*: This is the time period between the 2 and 6 minute marks during the tasks.
- *Reconsidered*: This is the time period between the 6 and 10 minute marks during the tasks, after the user has been reminded that they can change delay.
- *Total*: This is the total period during which the user is allowed to change their delays,

	Considered		Reconsidered		Total	
	Avg [mS]	StdDev [mS]	Avg [mS]	StdDev [mS]	Avg [mS]	StdDev [mS]
Nongreen	640.48	186.31	758.23	228.00	704.16	189.26
Green	642.44	186.44	775.66	218.45	708.56	186.99

Figure 5.5: The averages and standard deviations indicate that environmental prompting has no effect.

from the 2 to the 10 minute mark.

We had a total of 70 study subject, each of which used all 5 of the study applications. Half of the subjects were placed in the "green" population, the other half in the "nongreen" population, meaning we had the opportunity to collect up to 175 average values for each time period for each population. Due to some validation errors in the lab setup, we were only able to collect 165 values from the "green" population, and 146 values from the "nongreen" population. Figure 5.5 enumerates the averages and standard deviations of this collected data.

The results suggest that there is little to no difference in envelope settings between the green and nongreen populations. We further confirm this by running a T-Test and a Two One Sided T-Test (TOST)—with thresholds of 50ms and 100ms (corresponding to 5% and 10% of total delay, respectively) on the populations, results shown in Figure 5.6. We can see that for the Considered and Total data, all statistical tests say that there is no difference between the green and nongreen populations. For the Reconsidered data, a TOST cannot tell us with 95% confidence that these populations behave the same, and so it bears further looking into.

In designing the study, we wanted to ensure that subjects would not "forget" about the delay settings, and so added a reminder in the middle of the active part of the task. The careful reader will remember that this is the prompt at the 6 minute mark which

p-values	T-Test	TOST	
		thresh = 50	thresh = 100
Considered	0.71	**0.02**	**0.00**
Reconsidered	0.49	0.10	**0.00**
Total	0.84	**0.02**	**0.00**

Figure 5.6: Statistical tests indicate there is no effect of environmental prompting on any of the trace regions. Statistically significant results ($p < 0.05$) are in bold.

	Nongreen			Green		
	Average	Δms	T-Test	Average	Δms	T-Test
Considered	650.48	107.75	**0.00**	642.44	133.22	**0.00**
Reconsidered	758.23			775.66		

Figure 5.7: Once users have been reconsidered (reminded) of their ability to change the delay setting, they increase the amount.

delineates the Reconsidered and Considered regions. We were careful to try to not directly tell the user that they could *increase* their delay, but rather to remind the user that they could *control* their delay, and change it with the volume button interface. Figure 5.7 enumerates the averages of the Reconsidered and Considered regions for both green and nongreen populations, and also shows the p-value of a T-Test done to compare the results. We can see that there is a difference of more than 100ms in each population, with a confidence level of $>$ 99%. This suggests that, at least in an initial training phase, it would be beneficial to have reminders to users that they have the ability to control their delay settings.[4]

Applications In addition to analyzing the data for the entire set of users, we also ran statistical tests to compare the results for green vs nongreen users of each application.

[4]To be clear, this would be beneficial for both "parties" involved. It would be to the benefit of the overall system since it increases the amount of acceptable delay, and it would also be beneficial to the end-user, especially at times that they found the delay to be distracting. If the user were to forget that they could improve their performance, if perhaps only momentarily, they would simply get more and more frustrated with no recourse.

p-values	Considered			Reconsidered			Total		
	T-T	TOST (50)	TOST (100)	T-T	TOST (50)	TOST (100)	T-T	TOST (50)	TOST (100)
Douban	0.19	0.60	0.21	0.50	0.45	0.19	0.84	0.21	**0.04**
Kingsoft	0.92	0.16	**0.02**	0.58	0.31	0.06	0.72	0.22	**0.03**
RenRen	0.56	0.32	0.07	0.89	0.22	**0.05**	0.85	0.18	**0.02**
Weather	0.81	0.21	**0.03**	0.98	0.19	**0.03**	0.91	0.18	**0.03**
Youku	0.96	0.17	**0.03**	0.70	0.34	0.11	0.83	0.23	0.04

Figure 5.8: Once we segment the data into individual applications, the TOST does not indicate as strongly that green and nongreen behave the same, but the T-Tests do not indicate that they behave differently.

The results are presented in Figure 5.8. At this level of segmentation the TOST results are no longer strong enough to confidently say that the green and nongreen populations behave the same, however by the same token the T-Test results do not indicate that they behave differently, so the results seem to indicate there are no real differences for any of the applications.

Max by interval

As mentioned at the beginning of Section 5.4, there is not necessarily one simple way to establish a delay tolerance envelope from the study results. In addition to the area under the curve method, we evaluate the results of the study using what we call a *Maximum by interval* method. The basic idea behind this method is that—during the task duration, each subject is adjusting the delay setting as they accomplish the task, and so rather than averaging the setting over the duration, we look at the maximum delay setting held for at least X seconds, for varying values of X. An example of this is shown in Figure 5.9.

The results of the analysis are presented in Figure 5.10. The results of the max for interval analysis confirm our findings from the area under the curve analysis—when analyzing the two sample populations via both T-Test and TOST, we find no statistically significant differences between green and nongreen subjects.

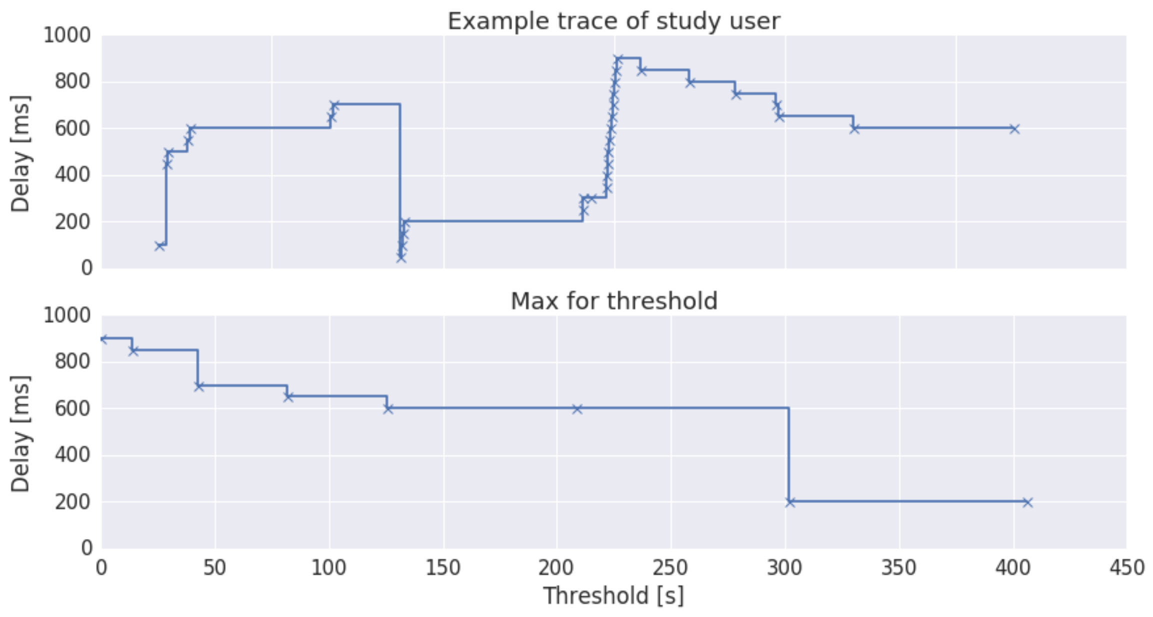


Figure 5.9: The top graph displays a delay setting trace from a single application an user, the bottom graph shows the maximum delay setting for that trace.

Threshold [s]	T-Test	TOST (thresh = 50)	TOST (thresh = 100)
0	0.59	0.11	**0.00**
5	0.78	**0.00**	**0.00**
10	0.72	**0.01**	**0.00**
20	0.70	**0.01**	**0.00**
30	0.66	**0.02**	**0.00**
60	0.55	0.06	**0.00**
120	0.77	**0.05**	**0.00**

Figure 5.10: Comparing green vs nongreen populations using the “max by interval method, TOST still shows that the populations behave the same.

Applications When we run the comparison for each application using the maximum by interval method, we find largely the same results as the area under the curve method. Results are enumerated in Figure 5.11.

Last level

As a last method of evaluating the user study, we consider the last value that a subject left the delay at. The idea here is that, after taking time to adjust their settings, and being reminded of their ability to control their delay, the last setting could be indicative of the subject's "true" delay tolerance envelope.

Average Delay Using the last level method, we find a somewhat higher delay tolerance envelope, but are ultimately unable to say with enough confidence that there is any difference between the green and the nongreen populations. The results of this analysis are presented in Figure 5.12.

Applications When we split the analysis of the populations into the different applications, the last level approach is unable to tell is confidently that the green and nongreen populations behave either different or the same. The results of this analysis are presented in Figure 5.13.

5.5 Conclusion

Our purpose in conducting this study was twofold:

p-values	T-T	TOST (50)	TOST (100)
	Interval = 0		
Douban	0.33	0.60	0.32
Kingsoft	0.52	0.37	0.10
RenRen	0.87	0.28	0.10
Weather	0.44	0.48	0.19
Youku	0.85	0.28	0.09
	Interval = 5		
Douban	0.79	0.21	**0.03**
Kingsoft	0.19	0.56	0.16
RenRen	0.84	0.09	**0.00**
Weather	0.55	0.26	**0.03**
Youku	0.87	0.12	**0.01**
	Interval = 10		
Douban	0.80	0.23	**0.05**
Kingsoft	0.35	0.44	0.11
RenRen	0.39	0.27	**0.02**
Weather	0.58	0.24	**0.03**
Youku	0.87	0.12	**0.01**
	Interval = 20		
Douban	0.78	0.24	**0.05**
Kingsoft	0.55	0.31	0.06
RenRen	0.57	0.22	**0.02**
Weather	0.72	0.19	**0.02**
Youku	0.78	0.20	**0.03**
	Interval = 30		
Douban	0.89	0.20	**0.04**
Kingsoft	0.50	0.35	0.07
RenRen	0.77	0.17	**0.01**
Weather	0.60	0.25	**0.03**
Youku	0.78	0.20	**0.03**
	Interval = 60		
Douban	0.68	0.32	0.09
Kingsoft	0.55	0.33	0.07
RenRen	0.65	0.30	0.07
Weather	0.63	0.30	0.07
Youku	0.80	0.26	0.06
	Interval = 120		
Douban	0.57	0.40	0.14
Kingsoft	0.54	0.36	0.09
RenRen	0.95	0.22	**0.05**
Weather	0.74	0.27	0.06
Youku	0.80	0.29	0.08

Figure 5.11: Using the max by interval method and decomposing results for each application, we can also no longer confidently say green and nongreen populations behave the same, but the likelihood of them being different from T-Test results is lower.

	Average [ms]	Std Dev
Nongreen	742.77	258.20
Green	788.15	242.62
T-Test	**TOST (t = 50)**	**TOST (t = 100)**
0.09	0.43	**0.02**

Figure 5.12: Last level analysis does not indicate lack of power of environmental prompting as strongly, but also does not confidently indicate the existence thereof.

p-values	T-T	TOST (50)	TOST (100)
Douban	0.11	0.79	0.50
Kingsoft	0.23	0.55	0.17
RenRen	0.52	0.47	0.21
Weather	0.85	0.22	**0.04**
Youku	0.54	0.44	0.18

Figure 5.13: Evaluating applications under the last level approach unable to confidently say green and nongreen populations behave the same.

1. To attempt to replicate the delay tolerance envelope we saw in the initial US study of Chapter 3, and to give more robust evidence for the values of this envelope by giving users direct control over their delay settings.

2. To analyze the effect of "environmental prompting" on the delay tolerance envelope of users—that is, the effect of telling users that if they are willing to accept more delay, their actions become more environmentally friendly / sustainable.

. In order to evaluate these avenues, we designed and conducted a user study in which subjects were asked to complete tasks in 5 popular Android applications, and asked them to set the delay on the Android smartphone as high as they would be comfortable with. Subjects were separated into two populations—one with environmental prompting and one without. We compared the envelopes of the populations via 3 different methods.

With regard to the delay tolerance envelope, we found levels that were very similar to the results of the initial in-the-wild study of Chapter 3, where we saw that there was no statistically significant difference in the satisfaction level of users for delays up to 750ms. Depending on which which method was used to calculate the envelope of a user, we found that the average envelope for subjects of this study varied between 650-780ms.

A far more surprising result was that we found no statistically significant difference between the green and nongreen populations when comparing all of the collected data. When we broke down the data into per-application based results, we were not able to say with $>$ 95% confidence that the populations behaved the same, however we were also not able to say that the populations were different.

Upon the conclusion of the analysis of this study, we arrived at two hypotheses as to why the environmental prompting had no effect on the users of this study.

1. The lack of difference could be attributed to cultural differences between the United States and China.
2. The lack of difference could be attributed to the fact that there is no virtue signaling of any kind present in the study. It could be the case that a person is only motivated to alter their behavior in a performance-degrading manner if their actions (and, implicitly, the "goodness" of their actions) are visible to the world around them.

In order to evaluate both of these hypotheses, we designed and conducted additional user studies, which are described in the following chapters.

Chapter 6

Is environmental prompting culturally dependent?

In this section, I describe work undertaken in studying how effective (if at all) environmental prompting is in altering the delay tolerance envelope of mobile users. More specifically, in the study of the previous chapter we found that there was no difference in green (prompted) vs nongreen (unprompted) study subjects of a lab-based study conducted in Beijing, China. One of our hypotheses for this lack of effect was that it could come down to cultural differences. Since the initial in-the-wild study conducted at Northwestern University did not include an environmental prompting component, we designed and conducted an IRB-approved user study to test the effect of environmental prompting on a US-based population.

In this study we created videos of Pinterest, a popular Android application, being used for 30 seconds with differing levels of delay being injected into the application for each video. We created a web-based survey in which subjects were shown each of the videos in a random order, with no indication as to the level of delay, and asked to rate how satisfied they would be with the performance of the application for each of the videos. We were able to collect data from 200 participants using the Amazon Mechanical Turk (MTurk) online crowdsourcing platform, and present the results below. As a validation of both the efficacy of MTurk as a source of data, and of the statistical power of the data collected, we also ran the study on 100 participants in China using the SoJump online crowdsourcing platform. This allows us to perform a direct comparison of the effect of environmental prompting on both cultural populations, rather than inferring differences or similarities between this study and that of Chapter 5.

6.1 Mechanical Turk

Amazon Mechanical Turk (MTurk) is an online crowdsourcing platform, designed to allow researchers, companies, and others to recruit online participants (known as "turkers") to accomplish paid tasks, such as filling out surveys, opinion polls, cognitive psychological studies, perform identification tasks, etc. MTurk allows researchers to advertise their tasks (known as "HIT"s[1] to turkers, showing both a description of the task as well as the monetary reward for successfully completing that task.

When a researcher creates their HIT, they choose how many participants are desired for a given batch, and transfer funds to the MTurk system, which then makes the HIT visible and available to turkers. Once a turker has completed their task, their results, identified by a unique and anonymous Worker ID, are made available to the researcher coordinating the HIT. The researcher is then able to approve or reject each individual submission, based on the sanity and validation of results. Once a turker's submission has

[1]Human Intelligence Tasks

been approved, the monetary funds are transferred directly to the turkers account.

MTurk has been used often to conduct psychological studies, such as [18, 88, 89], as well as in Human Computer Interaction [110] and in studying energy implications on mobile devices [56]. The strength of the MTurk platform is that it allows a researcher to advertise to a huge pool of potential participants, granting the ability to do large-scale studies in a short time-frame, as well as lowering the costs of motivating any individual user.

6.2 SoJump

Mturk provides the ability to filter participants based on various features, and since we wanted to evaluate the effect of environmental prompting on a US-based population, the one that we were interested in was their geographic location. Based on self-reported demographics of turkers [69], at least 70% of the participant pool was US-based. However, for our validation on a China-based population, we encountered difficulties in recruiting users via the MTurk platform.

In order to recruit from a Chinese-based population, we decided to advertise our study via the SoJump online crowdsourcing platform. SoJump is organized in much the same way as MTurk, however the advertisement process is not a portion of the platform. Study participants were advertised to via message boards at various Chinese academic institutions, and so participants were almost exclusively college students. As with the Peking University study of Chapter 5, there is no IRB approval process China, and so this study was conducted under the same protections and review of the US study.

6.3 Application augmentation via DPartner

In order to produce a controllably-delayed application, we once again turned to the segmentation framework of DPartner, full details on DPartner can be found in Section 3.1. The major differences in the usage of Dpartner for this study are:

- Since users will be reporting their satisfaction via a web-based application, we remove FBCollector—the visual frontend to the framework.
- Since we require direct control over the delay via a non-visible manner, we remove the random setting of delay, and instead have Dpartner read the desired delay from a configuration file placed on the smartphone upon application launch.

Videos of user interaction with the application are captured using the AZ Screen Recorder application [59], which makes user interactions (touching the screen, scrolling, etc) visible via a translucent white circle overlayed on the application. This allows a subject that is watching the video to know exactly when user interactions with the application occur, and lets them understand the delays that are occurring during the duration of the test case.

6.4 Study implementation and hosting

In MTurk, researchers have the ability to create their HIT either directly in the MTurk system, using basic forms, or to provide a link to a standalone survey, and have the participant conclude the HIT by inputting a validation code in the task that is provided to them from the linked survey. As a means of allowing more complicated interactions, ease of portability, and ensuring that subjects had no way to "cheat the system" and skip tasks, we elected to implement the survey as a standalone application. The survey was written in the Python Flask framework [105], and was hosted on the Amazon Web Services Elastic Beanstalk [6] (AWS EB) platform, which allowed for rapid deployments and hosting of the survey[2]. We used an EB instance located in Oregon.

For the China study, we use the same Flask-based application, running on an AWS EB instance, however for this study the instance was located in Tokyo, to reduce latencies for loading the videos.

6.5 User study

The goal of this study was to evaluate the hypothesis that the underlying reason behind a lack of changed delay tolerance envelope from environmental testing in the preceding study had been one of cultural context. Put plainly, we wanted to check if perhaps environmental prompting was appropriate for one study population but not the other one. To this end, we once again leveraged the Dpartner framework to create a study in which subjects would watch videos of a normal interaction with a popular application, and rate their perceived satisfaction of the performance.

Application

We chose Pinterest as the application for this study for a variety of reasons—Pinterest is an extremely popular website and its application is among the most popular free applications available on the Google Play application market, meaning that finding users familiar with its concept and operation would not be difficult. In addition, the common usage

[2]There is no requirement to use an Amazon.com owned host for MTurk studies, this simply happened to be the most convenient host for our purposes.

pattern of the application involves a lot of direct user interaction that causes additional requests to be made to the datacenter backend, which in turn prompts more interaction, so injected delays would be visible often and thus more likely to influence the satisfaction of a test subject. We also had experience augmenting this application in our previous study, and knew that we could do so with relative ease.

Subjects

Our study was IRB approved[3], allowing us to recruit users from the entire US and Chinese crowdsourced participant pools. We advertised the study to participants via the provided framework task discovery mechanism, which allowed potential participants to pick tasks based on description, duration, and compensation. Our selection criteria for each study was that participants had to be geographically located either in the United States or China, respectively for each study. We released the study availability in batches of 20, to both ensure that the study server would not become overloaded, and to take advantage of the popularity of newly published tasks. Participant slots were doled out in a "first come first served" fashion, and submissions would be validated manually. If a submission did not pass the validation check, it would be rejected, and that "slot" would be made available to the entire participant pool once again. As part of the study, each participant was also asked to fill out a short demographic survey, and a questionnaire about their familiarity with the application, smartphones in general, etc. Each validated participant was rewarded with $0.50 credited directly into their account.

[3]Northwestern IRB Project Number STU00093881

Test Cases

The test cases for this study were designed to mimic those of the previous study. We recorded a standard interaction with the Pinterest application—start on the home screen of the application, scroll down to see more content, click on an item to view more details, scroll down to view related content, click on one of the related pieces of content, navigate back to the main page and scroll a little more. Each interaction forces subsequent requests to be issued by the application for more content. The entire recording lasts 30 seconds. We recorded 5 videos, containing network requests delayed by 0, 250, 500, 750, and 1000ms. Each of these videos would become one test case in the study. The user would have no indication how much (if any) delay was being used in any given video, and care was taken to restrict the content in the video to be inoffensive.

Methodology

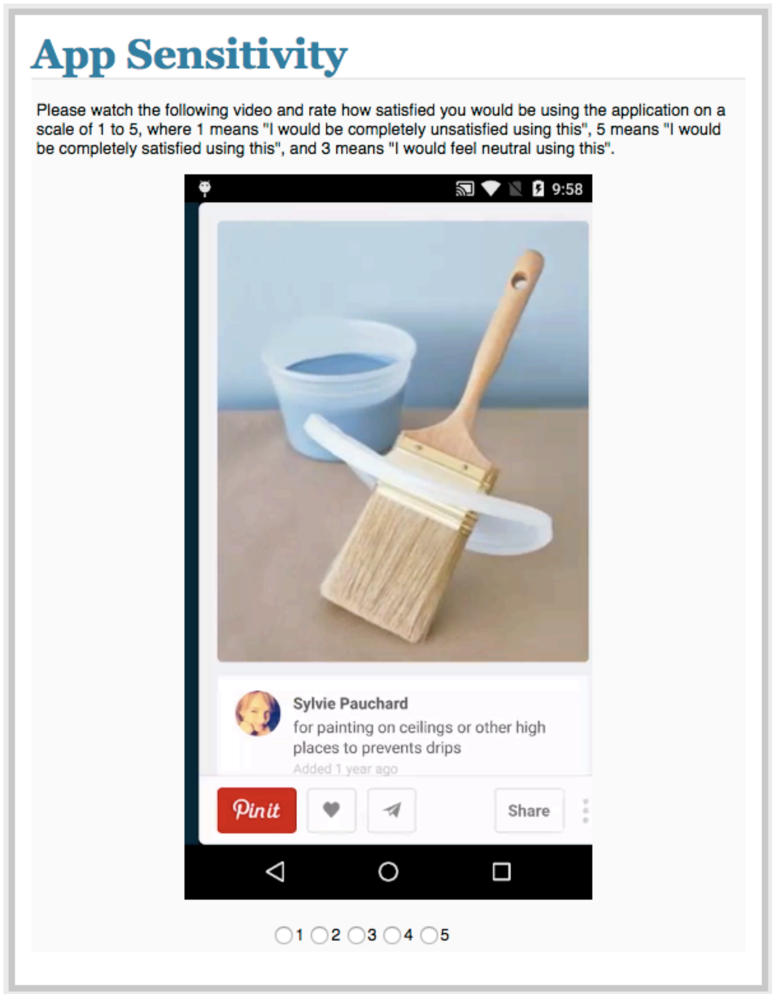
The subject would use their own personal computer for the study, and navigate between the MTurk system and EB hosted survey as necessary. All logs would be kept as a database on the EB instance, and verification codes would be submitted and kept in the MTurk management interface. The study would take approximately 7 or 8 minutes to complete.

The subject would begin the survey by following the link provided to them in the HIT description, and be asked to read and agree to the provided IRB consent form. The subject could print out the form, or request a separate copy via email. Since the entire process was carried out online, the users acceptance was recorded in lieu of a signature. After accepting the terms, the subject would be asked to answer a short questionnaire about their demographic information, as well as ranking their familiarity with the Pinterest application as well as smartphones in general.

At this point the subject would be provided with instructions on how they would complete the survey. The language of the instructions was identical for all users with one exception: whether or not it contained the environmental prompting. Users would be randomly chosen to be prompted or not, and if so chosen their instruction would contain the following prompt: "We have modified the application to behave in a more sustainable way, which means that it may seem a little slower, however this means that the cloud that is storing Pinterest data is also using less energy when this version is being used as opposed to the normal version."

The subject would be instructed to watch each of the videos, and rate how satisfied they would be using the application on a scale of 1..5, whose intent was described as "... 1 mean[ing] "I would be completely unsatisfied using this", 5 mean[ing] 'I would be completely satisfied using this', and 3 mean[ing] 'I would feel neutral using this'." The rating interface, which can be seen in Figure 6.1, would only appear once the video had been watched all the way through, thus ensuring that a subject could not simply skip watching any of the videos. The order of the videos would be chosen randomly for each user, to avoid any potential ordering effects.

Once the subject had completed watching and rating all of the videos, they would be asked to fill out an exit questionnaire, which provided them with a free-form response to describe what sort of annoyances they regularly experienced with smartphones.


App Sensitivity

Please watch the following video and rate how satisfied you would be using the application on a scale of 1 to 5, where 1 means "I would be completely unsatisfied using this", 5 means "I would be completely satisfied using this", and 3 means "I would feel neutral using this".

9:58
Sylvie Pauchard
for painting on ceilings or other high places to prevents drips
Added 1 year ago
Pin it
Share
1 2 3 4 5

Figure 6.1: Rating interface provided to the test subject. The radio buttons and "Rate" button would only be made visible once the video had completely played

6.6 US study results

This study involved 200 subjects, each of whom provided a rating for every one of the 5 videos they watched. In total, the study produced 1000 ratings, with 485 ratings belonging to the green (environmental prompted) population, and 515 belonging to the nongreen population. The demographics of the study subjects were:

- 62 were enrolled in a College or University, 138 were not.

Delay [ms]	Average [rating]	T-Test	TOST (t = 0.5)
0	4.02		
250	3.81	**0.02**	**0.00**
500	3.56	**0.00**	0.32
750	3.47	**0.00**	0.70
1000	3.08	**0.00**	1.00

Figure 6.2: Averages and statistical tests for US study results, delay noticeably affects satisfaction.

- 120 were male, 80 were female.
- 97 were chosen to be green, 103 were chose to be nongreen.
- 51 were between the ages of 18 to 25, 90 were between 26 and 35, 39 were between 36 and 45, 11 were between 46 and 55, and 9 were 56 or older.

Since we have a fair number of users representing various demographic categories, we can also explore if factors outside environmental prompting, such as age, gender, or enrollment in a University could have any effect on the ratings of users. The satisfaction averages, as well as statistical comparisons to the ratings of no delay, are presented in Figure 6.2. As mentioned earlier, we purposefully picked an application where injected delays would be readily noticeable, and the T-Test results show that this is indeed the case.

Environmental prompting has no effect on ratings

For each of the analyses of both the MTurk and China studies, we conducted a variety of tests for the corresponding populations we were comparing. To test the effect of the environmental prompting on the ratings, we calculate the average and standard deviation, and run both T-Test and TOST tests and present their p-values. However we are also interested in trying to see if we can find any trends that are not captured simply by the average of ratings. To this end we also calculate the skew, kurtosis, and median absolute deviation of each population.

Median absolute deviation (MAD) is a statistic that is designed to capture the variability of a dataset, much like the standard deviation, but because it lacks the squared component of the standard deviation, tends to be more robust to outliers [83]. The MAD is defined as

$$MAD([X_1, X_2, \ldots, X_n]) = median(|X_i - median(X)|)$$

Skew and kurtosis are the third and fourth moments of a population, respectively. The skewness of a population is a measurement of its asymmetry—when looking at the probability density function (PDF) of a population, a negative skew indicates that the tail to the left of the mean is either longer or fatter than that to the right, and a positive skew indicates the opposite. The kurtosis of a population is a measure of its "tailedness"—how likely outliers to the mean are. Kurtosis values are defined in relation to a normal distribution, which has a kurtosis of 3. Smaller values indicate that there are fewer outliers, larger values indicate that there are more outliers.

The result of of our analysis of the effect of environmental prompting are presented in Figure 6.3. We were expecting to see some effect from the prompting, however we can see from both the T-Test and TOST results that, at every delay level, there is no statistically significant difference between the green and nongreen populations! To contextualize the skew and kurtosis, we present histograms of each population for each delay level in Figure 6.4. The kurtosis values do not indicate any consistent trends, although the skew values get closer to 0 (normal-like) for all non-zero delays, which could mean that some

	Nongreen					Green						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	4.01	0.85	-0.49	2.49	1.00	4.02	0.87	-0.60	2.64	1.00	0.93	**0.00**
250	3.82	0.94	-0.67	3.22	1.00	3.84	0.83	-0.32	2.54	1.00	0.88	**0.00**
500	3.52	1.10	-0.44	2.37	1.00	3.58	0.98	-0.25	2.31	1.00	0.72	**0.00**
750	3.53	1.28	-0.52	2.14	1.00	3.39	1.00	-0.23	2.29	1.00	0.38	**0.02**
1000	3.11	1.31	-0.15	1.76	1.00	3.06	1.16	-0.08	2.21	1.00	0.80	**0.01**

Figure 6.3: Comparing green vs nongreen populations for US study, environmental prompting has no effect at any delay level.

lower ratings have been "redistributed" to slightly higher ones. Since the average does not change, this is most likely explained by green-conscious outliers, that is, there are likely to be a small minority of users in a population that are affected by environmental prompting, but their presence is not enough to make a difference in the aggregate.

Possible effects of demographic factors

Given the slight change in kurtosis, or "tailedness" of the green vs nongreen populations, we additionally wanted to ascertain if differences in ratings existed between any of the demographic groups represented in our study population. The analyses conducted and presentation of results is identical to that of the previous section.

College / University enrollment The thought behind segmenting the study population by whether or not the participant is enrolled in a college / university or not is that the academic environment often exposes persons not only to new technologies, but also new ideas, especially ideas that may not have entered the mainstream culture yet. The results of this split are presented in Figures 6.5 and 6.6. Based on the TOST, we can say that for delays of 0 and 250s there is no difference on the rating between the two groups. For the remaining delays, the TOST cannot say they behave the same, but the T-Test can also not tell us that the two groups behave differently.

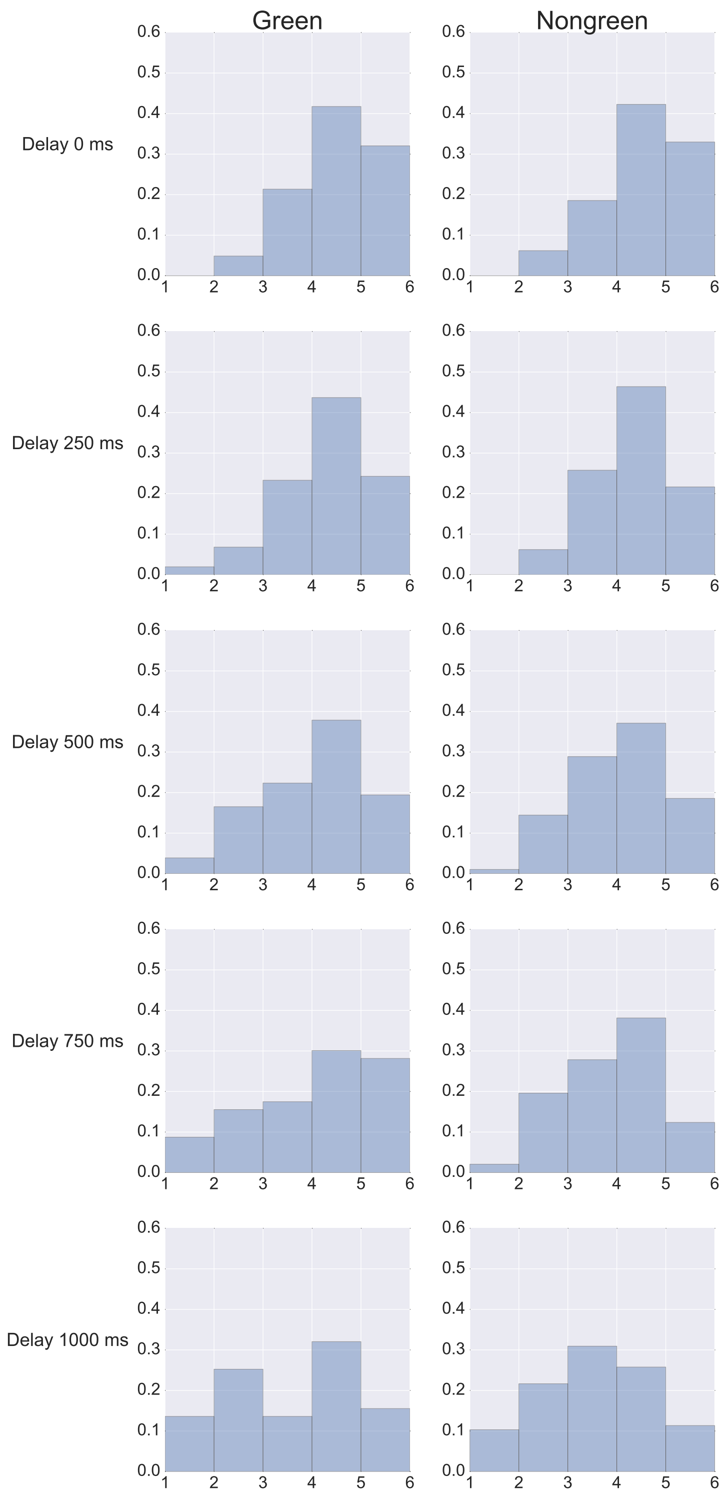


Figure 6.4: Distributions of ratings for green and nongreen subjects of US study.

	College/University					Not College/University						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	4.02	0.89	-0.72	2.87	1.00	4.01	0.85	-0.45	2.40	1.00	0.99	**0.00**
250	3.85	0.98	-0.83	3.57	1.00	3.81	0.85	-0.35	2.54	1.00	0.77	**0.00**
500	3.74	0.97	-0.54	2.25	1.00	3.46	1.06	-0.28	2.13	1.00	0.07	0.08
750	3.68	1.07	-0.58	2.60	1.00	3.37	1.18	-0.30	2.13	1.00	0.07	0.14
1000	3.29	1.32	-0.38	2.02	0.50	2.99	1.19	-0.01	1.97	1.00	0.14	0.14

Figure 6.5: Comparing College / University enrollment vs not for US study, no significant differences in the average but some distribution shape trends may emerge.

Looking at the shape and moments of the populations, the skew was more positive and the kurtosis was lower for people not enrolled in a college or university, which is suggestive of the fact that there may be a small population of college students who are more sensitive to the performance of mobile applications, but once again this effect is too small to be felt in the aggregate.

Male vs female Our study population had a (somewhat) surprisingly even split of male and female participants, so we decided to segment the data on gender and see if there were any differences between the two. The results are presented in Figures 6.7 and 6.8. Based on the T-Test and TOST, we can say that the ratings of the two populations behaved the same for all delays except 1000ms, where the TOST result does not have enough confidence. Looking at the moments, for all but 250ms delay the female population had a more negative skew, but there was no consistent trend for changes in kurtosis. This might be indicative of female participants being more sensitive to higher delays, but with the varying indication of outliers (from the kurtosis) we cannot safely assume this.

Age groups Given the assumption that younger people have a tendency to be more sensitive to changes in performance[4] we wanted to see if we could find any differences in the age groups of our study population. Based on the demographics, we had enough

[4]Ask any older person and they will happily tell you young people have no patience these days...

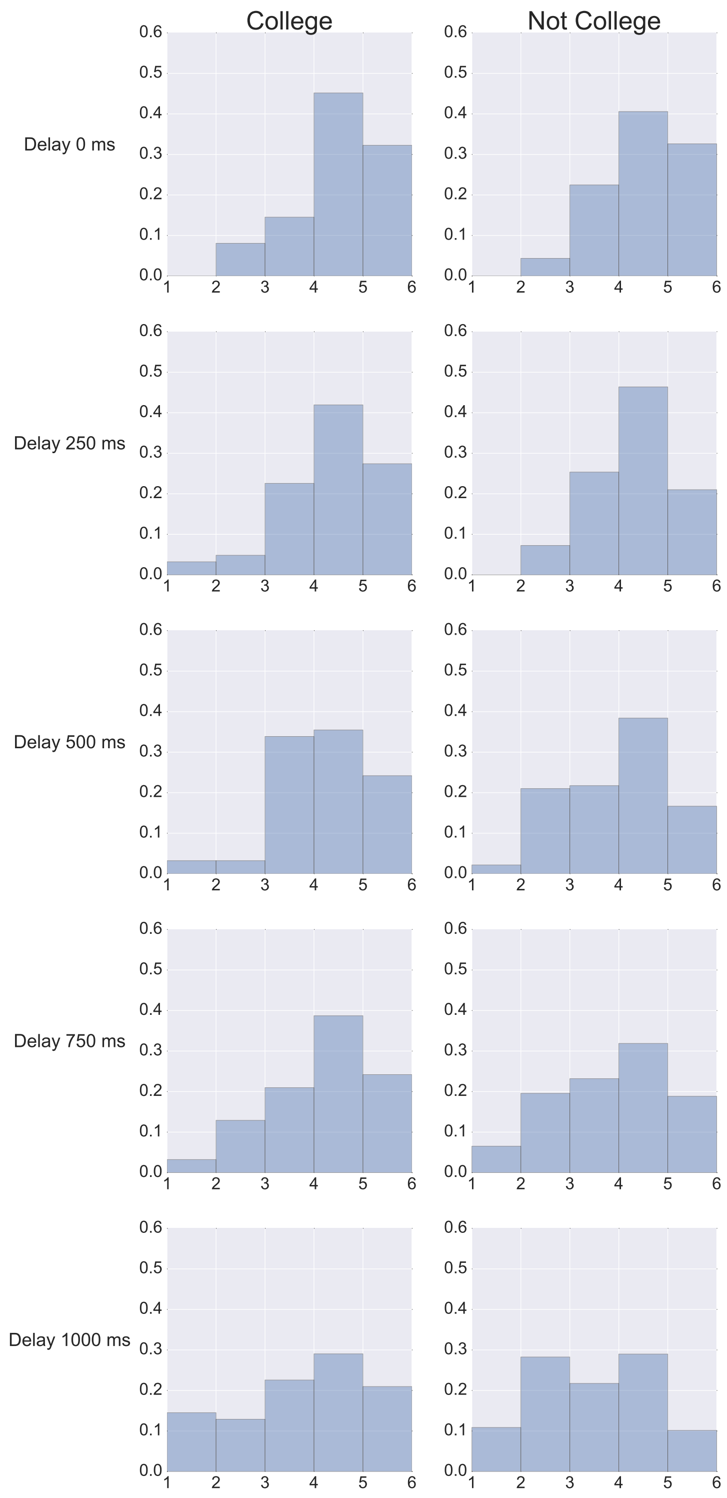


Figure 6.6: Distributions of ratings for collegiate and non-collegiate subjects of US study.

	Male					Female						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.96	0.84	-0.43	2.52	1.00	4.10	0.89	-0.73	2.74	1.00	0.26	**0.00**
250	3.78	0.89	-0.71	3.53	1.00	3.89	0.89	-0.30	2.20	1.00	0.42	**0.00**
500	3.46	1.02	-0.34	2.58	1.00	3.69	1.07	-0.46	2.20	1.00	0.13	**0.04**
750	3.38	1.15	-0.30	2.20	1.00	3.60	1.16	-0.53	2.37	1.00	0.18	**0.05**
1000	2.97	1.23	-0.04	1.96	1.00	3.26	1.23	-0.23	1.97	1.00	0.10	0.13

Figure 6.7: Comparing the ratings of male vs female subjects for US study, no differences.

participants to compare people in the age ranges of 18 to 25, 26 to 35, and 36 to 45. The full results are presented in Figures 6.9,6.10,6.11,6.12,6.13, and 6.14. Based on the T-Test and TOST results, we generally can say there is no difference between the populations, although we have the least confidence in the comparison of participants 18 to 25 and 36 to 45.

Looking at the moments, there is no consistent change in the kurtosis, but for most cases the skew is less negative for the older group in the comparison. Combined with the lower confidence in the sameness of groups 18 to 25 and 36 to 45, this suggests that there may indeed be a decline sensitivity with age, but perhaps we do not have enough older users. After all, 36 to 45 is not very old!

6.7 China study results

This study involved 100 subjects, each of whom provided a rating for every one of the 5 videos they watched. In total, the study produced 1000 ratings, with 215 ratings belonging to the green (environmental prompted) population, and 285 belonging to the nongreen population. The demographics of the study users are:

- 91 were enrolled in a College or University, 9 were not.
- 50 were male, 50 were female.

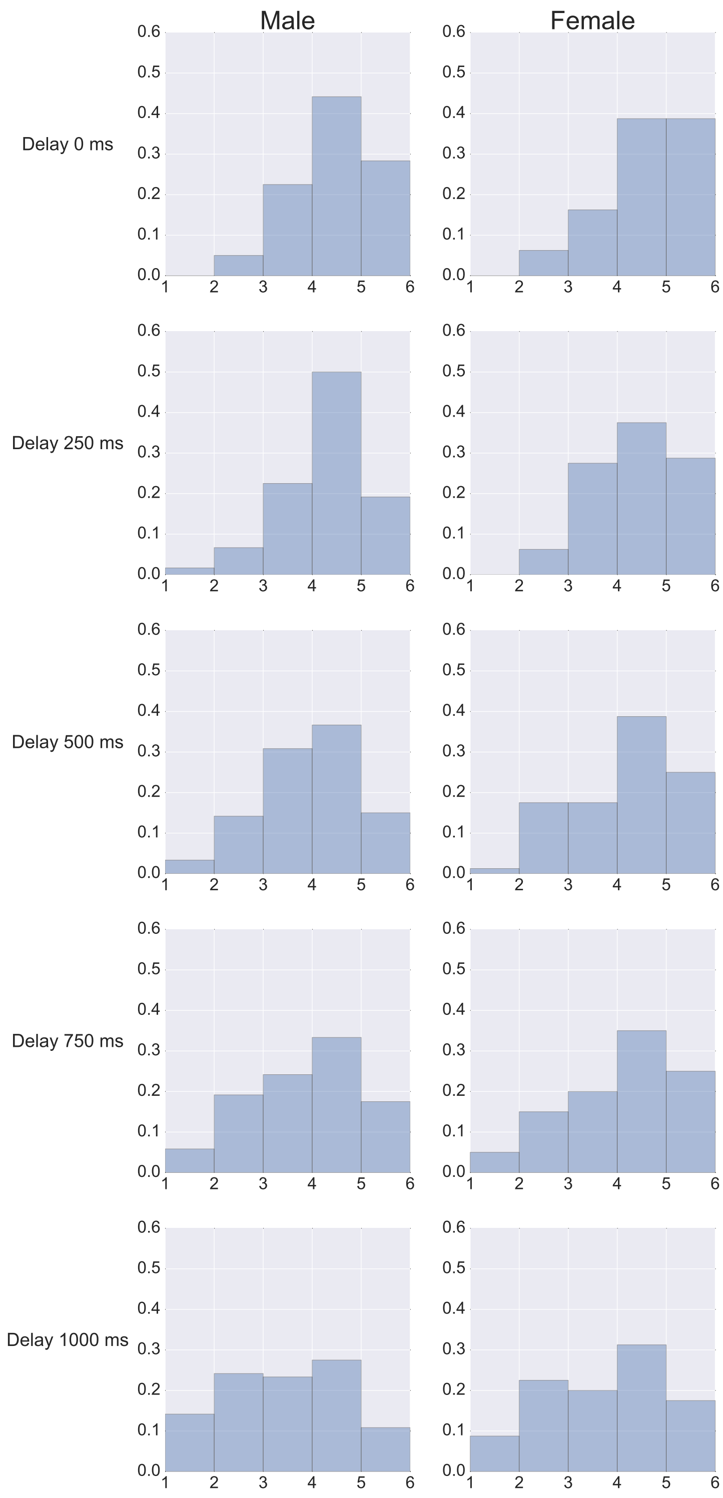


Figure 6.8: Distributions of ratings for male and female subjects of US study.

	18-25					26-35						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.84	0.92	-0.30	2.19	1.00	4.07	0.87	-0.54	2.41	1.00	0.16	**0.04**
250	3.80	0.97	-0.86	3.51	1.00	3.81	0.85	-0.38	2.55	1.00	0.97	**0.00**
500	3.73	1.01	-0.57	3.13	1.00	3.47	1.07	-0.30	2.13	1.00	0.16	0.10
750	3.51	1.18	-0.64	2.51	1.00	3.51	1.16	-0.39	2.16	1.00	0.99	**0.01**
1000	3.14	1.30	-0.26	1.96	1.00	3.14	1.21	-0.16	2.01	1.00	0.97	**0.01**

Figure 6.9: No strong differences between subjects of age ranges 18-25 and 26-25 of US study.

- 43 were chosen to be green, 57 were chose to be nongreen.
- 77 were between the ages of 18 to 25, 22 were between 26 and 35, 0 were between 36 and 45, 1 was between 46 and 55, and 0 were 56 or older.

Given the lower number of users, we are not able to segment based on as many demographic factors as in the US study. The satisfaction averages, as well as statistical comparisons to the ratings of no delay, are presented in Figure 6.15. Comparing these results to the US study, the average ratings for each delay amount are lower than their corresponding MTurk rating, but the trends remain the same—this study population also clearly perceives the degradation of performance.

As a reminder to the reader, the reason for conducting an additional analysis of a Chinese participant population is twofold: to "sanity check" that the lack of effect of environmental prompting in the Peking University study was correct, thus grounding our confidence in the results of a crowdsourced study, as well as providing additional support for the findings of the US study.

Environmental prompting has no effect on ratings

The results for the effect of environmental prompting on ratings are presented in Figures 6.16 and 6.17. When we compare the green and nongreen populations, we once

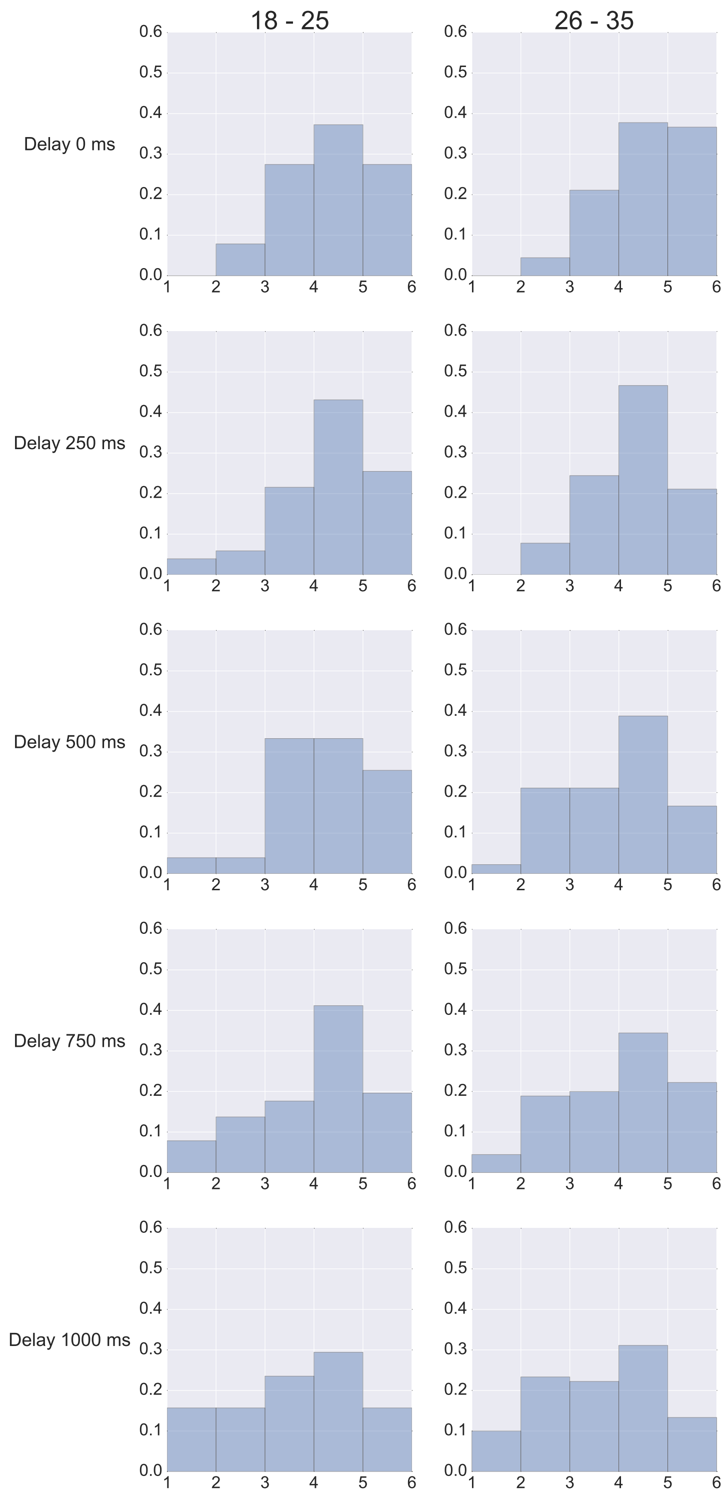


Figure 6.10: Distributions of ratings of 18-25 and 26-35 year old subjects of US study.

	18-25					36-45						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.84	0.92	-0.30	2.19	1.00	4.08	0.76	-0.82	3.81	0.00	0.20	0.08
250	3.80	1.01	-0.86	3.51	1.00	3.85	0.86	-0.18	2.17	1.00	0.83	**0.01**
500	3.73	1.01	-0.57	3.13	1.00	3.85	0.86	-0.35	2.28	1.00	0.55	**0.05**
750	3.51	1.18	-0.64	2.51	1.00	3.31	1.09	-0.16	2.35	1.00	0.41	0.11
1000	3.14	1.30	-0.26	1.96	1.00	3.08	1.29	0.07	1.79	1.00	0.83	0.06

Figure 6.11: Comparing subjects of age ranges 18-25 and 36-45 of US study, we are not able to confidently claim there is no difference in rating.

again find that the T-Test cannot tell us the the two populations behave differently for any delay setting, although the TOST result falls short of confidence for delays of 250 and 1000. Looking at the moments of the populations, there are no clear trends that emerge for either the skew or kurtosis. The differences in confidence here could be due to the smaller number of study participants.

Possible effects of demographic factors

We wanted to repeat the demographic analysis conducted in the MTurk study, however due to the smaller population size and academic participant pool, we were restricted to comparing male vs female participants, and the age groups of 18-25 vs 26-35.

Male vs female The results of comparing male and female populations are presented in Figures 6.18 and 6.19. Based on the T-Test and TOST results, we can say that there is no difference between the two populations. There are no clear trends in the changes of the skew or kurtosis, which points to there definitively being no differences in the perception of and tolerance for performance in this population.

Age groups The results of comparing the age groups of 18-25 and 26-35 are presented in Figures 6.20 and 6.21. Similarly to the MTurk results, the TOST results do not indicate that

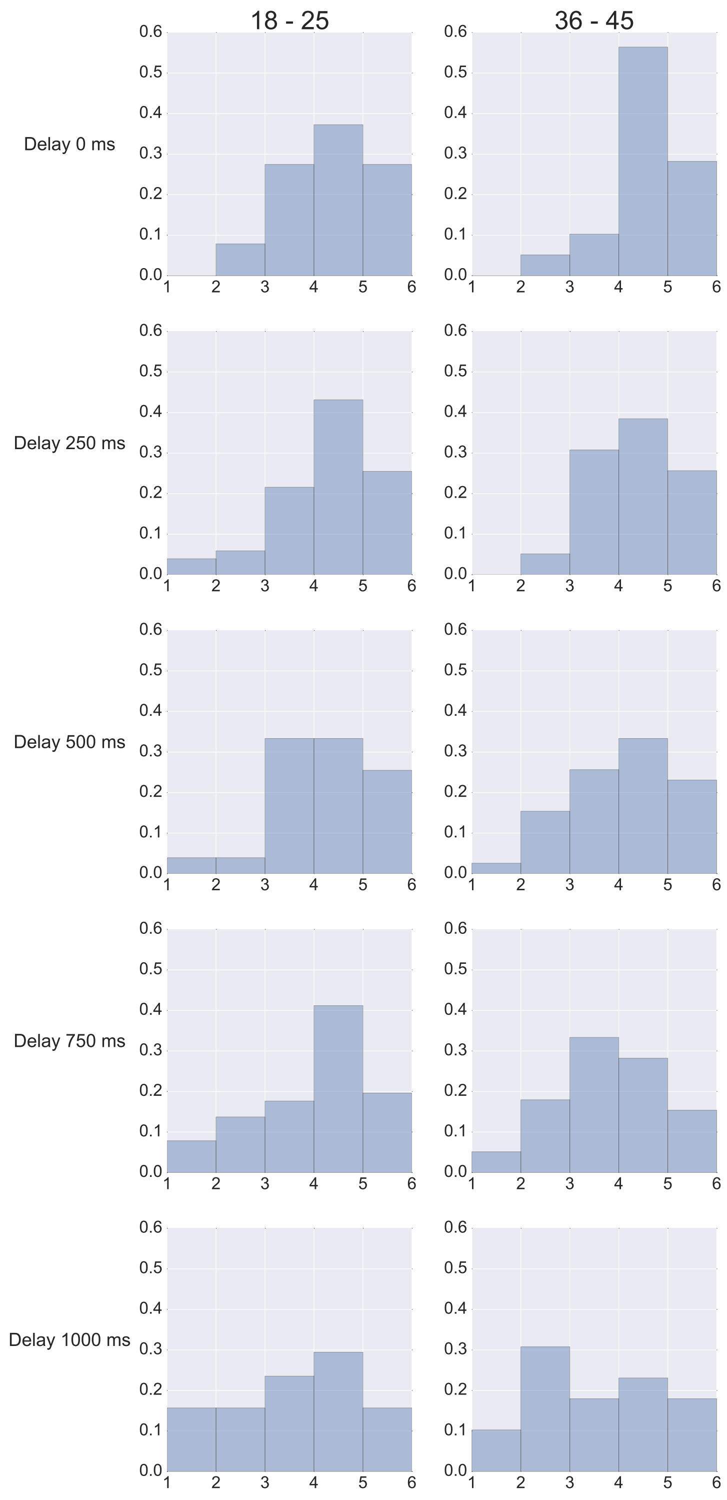


Figure 6.12: Distributions of ratings of 18-25 and 36-45 year old subjects of US study.

	26-35					36-45						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	4.07	0.87	-0.54	2.41	1.00	4.08	0.76	-0.82	3.81	0.00	0.95	**0.00**
250	3.81	0.85	-0.38	2.55	1.00	3.85	0.86	-0.18	2.17	1.00	0.83	**0.00**
500	3.47	1.07	-0.30	2.13	1.00	3.59	1.08	-0.35	2.28	1.00	0.56	**0.04**
750	3.51	1.16	-0.39	2.16	1.00	3.31	1.09	-0.16	2.35	1.00	0.35	0.09
1000	3.14	1.21	-0.16	2.01	1.00	3.08	1.29	0.07	1.79	1.00	0.78	**0.0**

Figure 6.13: No strong differences between subjects of age ranges 26-25 and 36-45 of US study.

the populations behave the same with a high enough confidence, but the T-Test results also do not indicate that they behave differently. The one interesting exception is the case of no delay, where the T-Test p value was 0.01, and the 26-35 group had ratings lower by 0.86 rating points.

6.8 Conclusions

In this chapter I described the work we undertook to answer the question: is the reason we did not see any effects of environmental prompting in our earlier PKU study that there are fundamental cultural differences in the perception of environmental friendliness among that study demographic. We designed a crowdsourced study in which participants watch and rate how satisfied they would be using a recorded application at various delay levels, and conducted it on two geographically distinct populations, one from the United States and one from China. The purpose of including a Chinese population was twofold: to provide a sanity check against the earlier PKU study, lending credence to the crowdsourcing approach, and in turn to strengthen the results from the US population.

In both study populations we found there there were no differences in rating that were attributable to the environmental prompting. We found that the means of prompted and unprompted users did not have a statistically significant difference, and we also did not

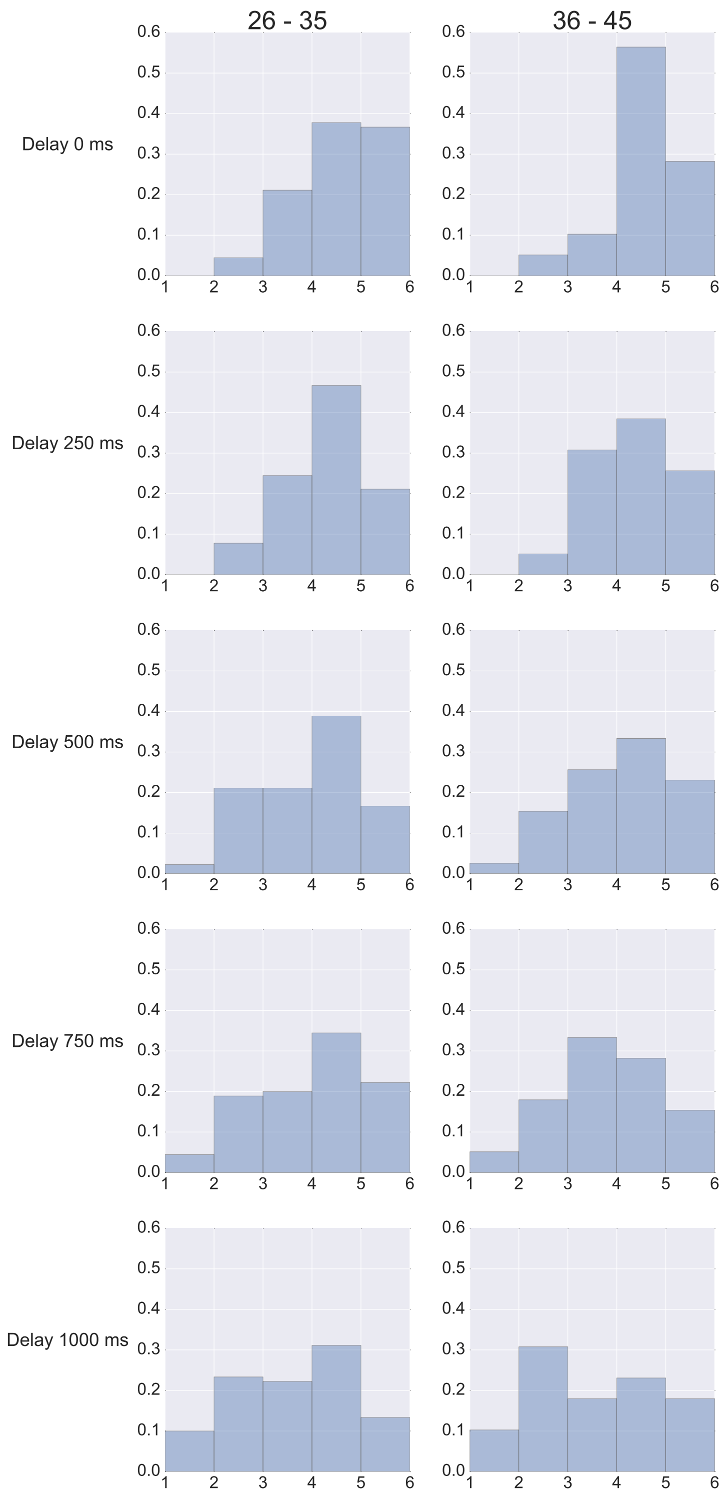


Figure 6.14: Distributions of ratings of 26-35 and 36-45 year old subjects of US study.

Delay [ms]	Average [rating]	T-Test	TOST (t = 0.5)
0	3.62		
250	3.21	**0.01**	0.28
500	3.00	**0.00**	0.79
750	2.98	**0.00**	0.82
1000	2.50	**0.00**	1.00

Figure 6.15: Averages and statistical tests for China study results, delay noticably affects satisfaction.

	Nongreen					Green						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.67	1.00	-0.57	2.65	1.00	3.56	1.17	-0.67	2.56	1.00	0.63	**0.04**
250	3.11	1.00	-0.32	2.53	1.00	3.35	1.10	-0.09	1.95	1.00	0.26	0.12
500	2.98	1.10	-0.04	2.12	1.00	3.02	0.93	0.30	2.64	1.00	0.84	**0.02**
750	3.04	1.12	0.08	2.36	1.00	2.91	1.07	0.19	2.15	1.00	0.57	**0.05**
1000	2.37	1.28	0.55	2.12	1.00	2.67	1.25	0.28	2.03	1.00	0.24	0.23

Figure 6.16: Comparing green vs nongreen subjects for China study, environmental prompting generally has no effect.

find any consistent trends in changes to the higher moments of the datasets when we looked at skew and kurtosis. In addition, we compared the distribution of ratings for various demographic splits such as collegiate enrollment, gender, and age group. Based on the population means, we also did not see any significant differences in rating for these splits, although we did notice some potential trends in the distribution changes via the higher moments. This variability strengthens the need for further research into the impact of prompting or demographics on an individual, and ultimately the need to consider the delay tolerance envelope of individual users.

Our findings in this chapter discard the hypothesis that differences in cultural context are responsible for the lack of efficacy of environmental prompting on the delay tolerance envelope of users. This leaves us with our alternate hypothesis—that users are more spurred on by a signaling mechanism, which rewards them in some way of their “good” actions, even if it is only via a comparison to their peer group or surroundings. We explore

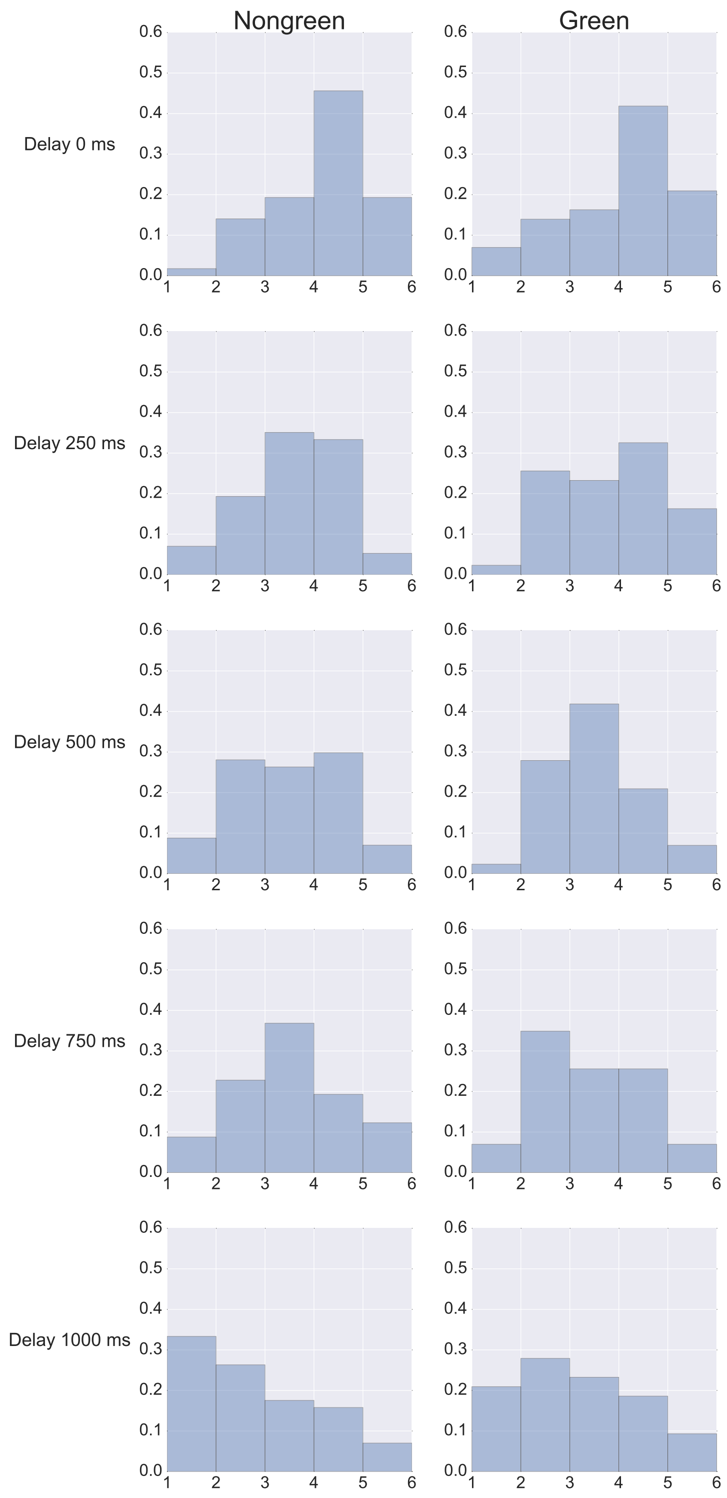


Figure 6.17: Distributions of ratings of green and nongreen subjects of China study.

	Male					Female						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.58	1.18	-0.52	2.35	1.00	3.66	0.95	-0.81	3.12	0.00	0.71	**0.03**
250	3.20	1.13	-0.15	2.33	1.00	3.22	0.97	-0.18	2.09	1.00	0.93	**0.01**
500	3.00	1.02	0.11	2.51	1.00	3.00	1.04	0.00	2.16	1.00	1.00	**0.01**
750	3.00	1.10	0.18	2.17	1.00	2.96	1.11	0.08	2.38	1.00	0.86	**0.02**
1000	2.54	1.22	0.34	2.03	1.00	2.46	1.33	0.49	2.06	1.00	0.76	**0.05**

Figure 6.18: Comparing male vs female subjects of China study, there is no difference.

the effects of having users compare their actions against those of their surrounding group via peer pressure mechanisms in the next chapter.

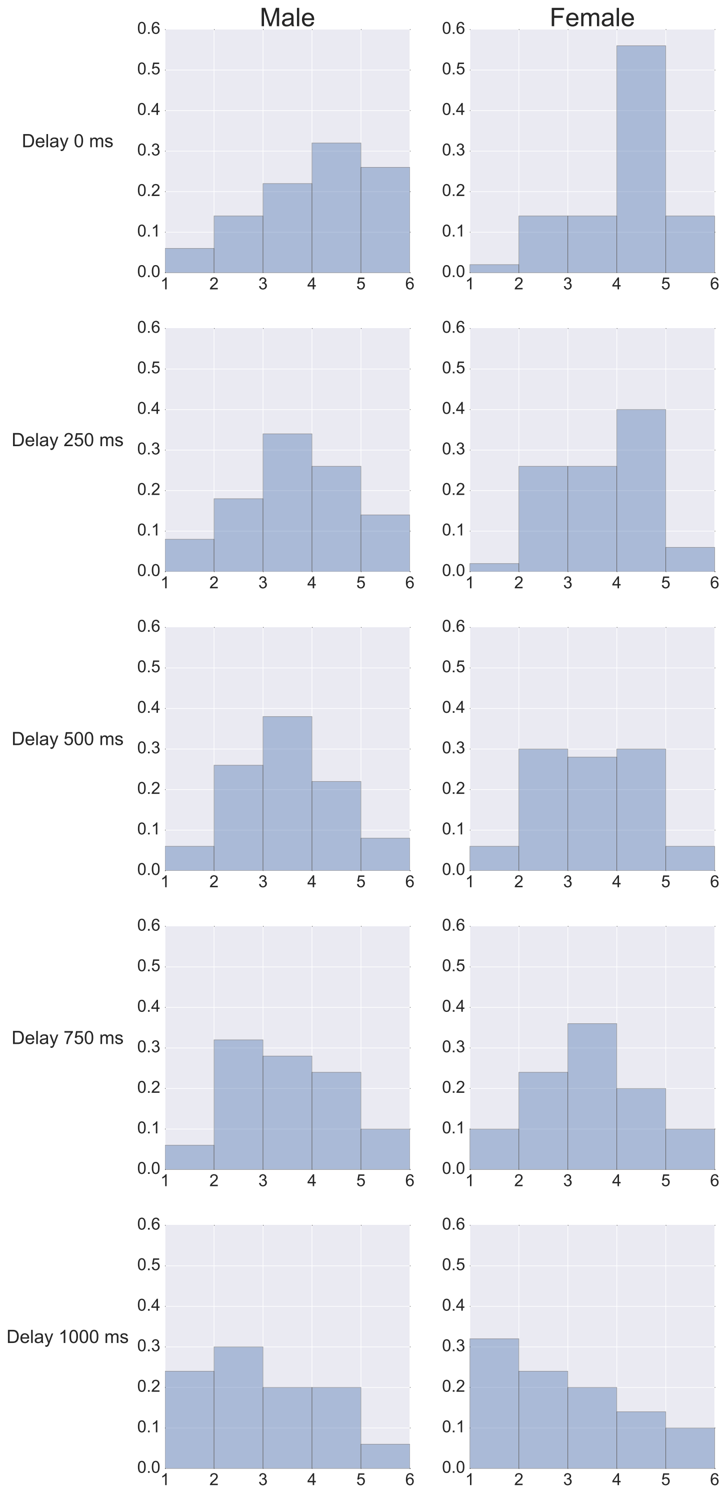


Figure 6.19: Distributions of ratings for male and female subjects of China study.

	18-25					26-35						
Delay	Avg	STDEV	Skew	Kurtosis	MAD	Avg	STDEV	Skew	Kurtosis	MAD	T-Test	TOST
0	3.81	0.95	-0.68	2.97	1.00	2.95	1.22	-0.06	1.87	1.00	**0.01**	0.92
250	3.31	1.00	-0.26	2.57	1.00	2.82	1.15	0.36	2.08	0.50	0.08	0.49
500	2.97	0.99	0.05	2.30	1.00	3.05	1.15	0.09	2.31	1.00	0.80	**0.05**
750	3.00	1.09	0.12	2.37	1.00	2.86	1.14	0.27	2.11	1.00	0.63	0.09
1000	2.39	1.22	0.47	2.13	1.00	2.77	1.35	0.19	1.86	1.00	0.25	0.35

Figure 6.20: Comparing subjects of age ranges 18-25 and 26-35 of China study, we cannot confidently say their ratings are the same, but also not that they are different.

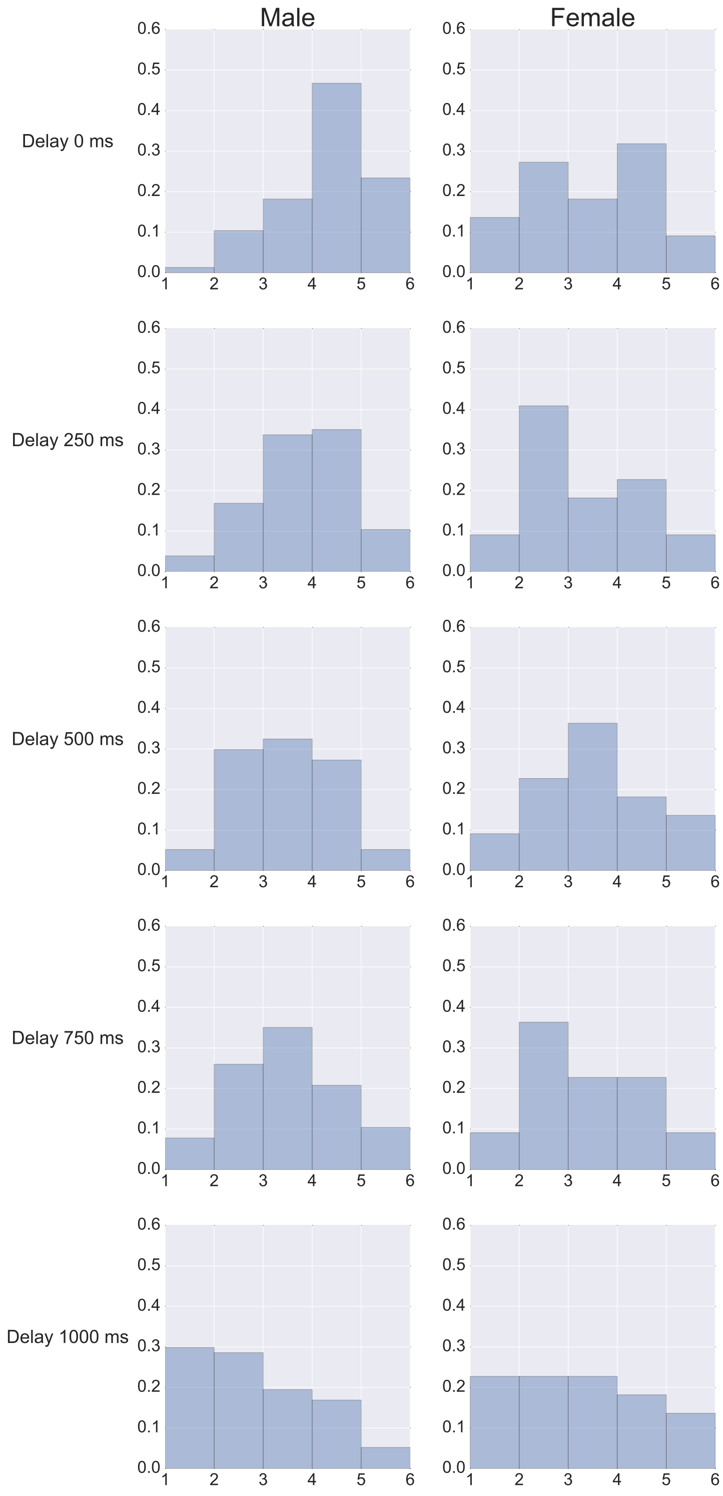


Figure 6.21: Distributions of ratings of 18-25 and 26-35 year old subjects of China study.

Chapter 7

What effect does peer pressure have on the delay tolerance envelope?

In this chapter, I describe work I did in investigating the effects of peer pressure on users delay tolerance envelope. I describe a user study that was done at Peking University as well as a study conducted via Amazon Mechanical Turk in the US.

As mentioned in Chapter 6, we established 2 hypotheses for the lack of effect of environmental prompting on the delay tolerance envelope, and had discarded the hypothesis of cultural differences. We then moved on to a lack of signaling as a potential cause—perhaps just the knowledge of being more environmentally friendly was not enough to change people's behavior, but instead what was needed was a way for a user's goodness to be in some way either visible to or comparable with the outside world.

Green signaling effects are not unknown in the scientific literature—perhaps the most notable modern example is that of the Toyota Prius. Sexton and Sexton describe what they dub *conspicuous conservatism*, "...in which individuals seek status through displays of austerity amid growing concern about environmental protection"[112]. In their study of consumer spending on automobiles, they find that consumers are willing to spend more money on the Prius over other cars which have equal or greater "greenness", and one finds that the Prius is disproportionately represented, despite its higher cost[1]. The authors attribute this to the green halo, or visible status that owning an automobile that signals its inherent goodness. So we see that, if moral status signaling is present, people are not only willing to behave differently, but even spend significantly more money to attain this signal.

Virtue signaling allows a user to broadcast the status of their moral virtue to the outside world, but perhaps we can capture an even simpler effect: that of peer pressure. Are users willing to alter their behavior if they are told that their virtue is significantly *different* than that of their surrounding group, even if their virtue is not directly broadcast to any of those members? Our investigation of the effect of peer pressure consists of two studies: we augmented the interface of the study conducted in Chapter 5 at Peking University, and also designed an interactive MTurk study using the same interface elements to run on a larger population. The idea behind the interface is to contextualize their current delay setting by showing a comparison of them versus the average of the whole population. The group averages presented are artificially chosen to be either very high or very low compared to the observed delays, in order to test participants responses to both stimuli.

The contribution of this work is:

- In a user study conducted on 20 participants using 5 applications on a mobile device in a controlled lab setting, we show that the inclusion of peer pressure causes subjects to respond to environmental prompting by accepting higher delays.

[1]The authors quantify the size of this cost difference with the cost of a Corolla hybrid, which has the closest feature-set to the Prius. They estimate the difference to be between $710 and $1925

- In this same study, we discard the possibility that subjects are simply "chasing" the group average by showing that subjects who are told they outperform the group do not have lower delay tolerances than subjects who had no peer pressure.
- In a user study conducted on 400 participants playing an online game via the Mechanical Turk crowdsourcing platform, we show that although environmental prompting does not produce an effect in this scenario, there is still a distinguishable difference on the delay settings of users who were told they are performing better and worse than the group average.

7.1 Peer Pressure interface

In the previous study of Chapter 3 we presented users with an interface that displayed to them one piece of information: their current delay setting. In order to leverage the effects of peer pressure, we would need to be able to show them a second datapoint, the collective behavior of their peer group, but perhaps more importantly we would need to allow the two pieces of data to be instantly and easily compared by the user. To this end, we created a simple interface that would simultaneously show to the user their controllable delay setting, as well as the simulated average of the study population. An example configuration is shown in Figure 7.1.

We chose 35% and 85% to be the low and high values for the group delay, respectively, as those were values that were outside of the normal averages, but still believable for a user. We also introduced a random nudge factor, such that the shown average could vary by 5% above or below the chosen average, which would be applied randomly over time to give the impression that the group average was changing, and make the user think that

Figure 7.1: The peer pressure interface presented to users. The red bar indicates the users current delay, while the green bar indicates the average of the group. This allows the users to anonymously compare themselves to the group at large.

it was indeed tracking a live population. The group average was displayed as a green “progress bar” in the overlay.

The user setting was displayed as a red bar floating on top of the group average bar, so that the user could see at all times both where their delay was and how it compared to the group. For the PKU study, the user would control their delay settings via the same mechanism described earlier in Chapter 3, using the volume keys. For the new MTurk study, the user was given keyboard controls that will explained in detail in Section 7.4.

7.2 PKU study

The purpose of this study was twofold—one, to establish if a peer pressuring element would cause people to care more about their environmental friendliness, and two, to test if users would be susceptible to groupthink mentality. What we mean by this is, if seeing a high group average increased users’ envelope up, would seeing lower group averages cause a drop in the user envelope? The applications chosen, subject recruitment, compensation, methodology, and testbed of this study is identical to the earlier PKU study of Chapter 3, with the only and notable exception of the peer pressuring interface replacing the notification bar interface described in the same Chapter. Each subject would be randomly placed into the leading (85% group average), or trailing (35% group average) category when they arrived.

Subjects

Peking University does not have a formalized IRB process, however we designed the study in a very similar way to the original US study, and made sure to take the same steps to preserve subject anonymity and not collect any personally identifying information. We advertised to the broader Peking University population via flyers posted in computer labs and University hallways / bulletin board, and email advertisements. Our selection criteria was that the subject had to be familiar with mobile phones, and have familiarity with at least some of the applications that they would use during the study. We selected the first 20 participants who responded and qualified. As part of the study, each subject would also fill out entrance and exit questionnaires, which included demographic information. At the end of the study, each participant was rewarded with a gift card worth 35 Chinese Yuan (approximately 5 USD). The demographics are enumerated in Figure 7.2.

7.3 PKU results

This study involved 20 subjects, and produced full results for each of the participants. For each subject, and each application, we collected the times of the events for each task (begin, 2 minute prompt, 6 minute prompt, end), the time and delay level of any time the user changed the application, which group delay population they had been assigned to, as well as properties about each network request the was sent by the study applications.

Users

Age	18-25	16
	25-35	4
Gender	Male	4
	Female	16
Area of Study	Computer Science	3
	Science / Engineering	4
	Medicine	3
	Law / Policy	5
	Other	5
Length of smartphone usage	0-1 Months	0
	1-6 Months	0
	6 Months - 1 Year	0
	1-2 Years	0
	2+ Years	20
Smartphone Type Owned	Android	10
	iOS	10
Carrier	China Mobile	12
	China Unicom	8

Figure 7.2: User study demographics.

In total, the study produced 2410 delay setting changes, which we use as the basis for our analysis. We use the same approaches from the previous study for evaluating each subjects delay tolerance envelope: area under the curve, maximum for interval, and last level.

Area under the curve

We had a total of 20 study subjects, each of which used all 5 of the study applications. Half of the subjects were placed in the "green" population, and the other half were placed in the "nongreen" population, giving us the opportunity to collect 50 average values for each population. The averages and standard deviations for all users are presented in Figure 7.3.

	Considered		Reconsidered		Total	
	Avg	StdDev	Avg	StdDev	Avg	StdDev
Nongreen	630.24	211.63	794.64	226.19	710.86	199.43
Green	696.12	106.44	862.85	132.14	778.82	110.78

Figure 7.3: Average value and standard deviation of the user delay envelope, as calculated via the Area Under the Curve Method.

	T-Test	TOST	
		thresh = 50	thresh = 100
Considered	**0.05**	0.68	0.16
Reconsidered	0.07	0.69	0.20
Total	**0.04**	0.71	0.16

Figure 7.4: P-value results of Statistical tests of whether the means of envelope for green and nongreen populations behave the same. Statistically significant results ($p < 0.05$) are in bold.

Comparing these results to the earlier study, we immediately see two effects emerge:

1. The difference in average between green and nongreen groups is fairly obvious, with the green population envelope consistently being 60ms larger than the nongreen envelope.

2. The standard deviation of the green envelope is greatly reduced, consistently being half as large as the standard deviation of the nongreen envelope.

We further verify these results by running T-Test and TOST analyses on the population datasets, results are shown in Figure 7.4. With the incorporation of the peer pressuring element, we can now see a clear statistical difference between the green and nongreen populations. It is also important to note that this difference is visible **regardless of whether the group average is higher or lower than the subjects setting!**

Due to the lower number of users in this study, we did not have enough data to further decompose the analysis of individual applications, and thus this is not presented in this

Threshold [s]	T-Test	TOST (thresh = 50)	TOST (thresh = 100)
0	0.35	0.42	0.09
5	0.35	0.12	**0.00**
10	0.30	0.15	**0.00**
20	0.14	0.32	**0.01**
30	0.09	0.50	**0.05**
60	0.13	0.49	0.06
120	0.25	0.45	0.08

Figure 7.5: Comparing green vs nongreen populations using the "max by interval method, differences are no longer clear.

Chapter.

Max by interval

When we compare the envelopes of the green and nongreen populations with the *maximum by interval* method, we no longer see clear statistical differences, at any of the intervals tested. The results of the analysis are shown in Figure 7.5. The analysis does not convincingly suggest that the populations behave the same, either, so it could simply be the case that there are not enough data points from this study. If we look at how often users changed their delay setting, we see that in the previous study there were an average of 111.14 changes per user whereas for this study there were an average of 120.5 changes per user, so the peer pressuring element could have caused the stability of the interval method to decrease, or the method could not be an appropriate way of calculating the envelope for any user. Further research and analysis is needed to determine the efficacy of this method.

	Average [ms]	Std Dev
Nongreen	825.00	224.77
Green	898.00	119.57

T-Test	TOST (t = 50)	TOST (t = 100)
0.05	0.73	0.23

Figure 7.6: Average delay setting, Standard Deviation, and statistical tests for the Last Level Analysis.

Last level

The results of analyzing the envelope using the last level method further support there being a difference between the green and nongreen populations, and are enumerated in Figure 7.6. We also see the same reduction in standard deviation, which adds confidence to both the utility of the area under the curve method, as well as our interpretation of the results of the results.

Leading and trailing

Study participants were randomly placed into either the leading or trailing categories, signifying that their presented group average would be at the high or low level, respectively. We have already seen that the inclusion of even such a simple peer pressuring mechanism induced the environmental prompting to be more effective, but we also wanted to verify this by showing that participants were not simply "chasing the group", by comparing the results of users in each group to the averages from the initial study.

The averages of the two groups are presented in Figure 7.7. We can clearly see the differences between the two groups, and we verify these differences with T-Tests—the p-values for the Considered, Reconsidered, and Total periods are **0.00**, **0.01**, and **0.01** respectively.

Leading	Considered		Reconsidered		Total	
	Avg	StdDev	Avg	StdDev	Avg	StdDev
Nongreen	546.86	203.91	756.72	234.51	660.22	199.72
Green	647.55	98.48	797.29	144.28	722.59	111.29

Trailing	Considered		Reconsidered		Total	
	Avg	StdDev	Avg	StdDev	Avg	StdDev
Nongreen	695.62	198.60	832.56	210.83	761.50	185.81
Green	744.70	90.69	928.40	74.23	835.05	76.39

Figure 7.7: Average value and standard deviation of the user delay envelope, as calculated via the Area Under the Curve Method, for trailing and leading

T-Test p-values	Considered	Reconsidered	Total
Nongreen	**0.02**	**0.00**	**0.00**
Green	**0.01**	**0.00**	**0.00**

Figure 7.8: Subjects in the trailing group had their averages increased.

Following the group? In order to figure out what effect the level of the group delay had on study participants, we compared the averages of each group with the averages of the 70 participants from the earlier study, which we consider to be the baseline. As a reminder to the reader, those earlier participants went through the same tasks on the same applications, with the same ability to control their delay, they were simply not given any kind of peer pressuring interface.

The comparison between subjects in the trailing group and the baseline is presented in Figure 7.8. Based on the T-Test and the averages, it is clear that presenting subjects with a high group average has increased up their delay tolerance envelope. Turning to the leading group, we enumerate the comparison between the group and the baseline in Figure 7.9. Looking at the results of the statistical tests and the averages, we can say that having a low group average did not lower the delay tolerance envelope.

T-Test p-values	Considered	Reconsidered	Total
Nongreen	0.14	0.60	0.67
Green	0.22	0.97	0.56
TOST p-values (t = 50)	Considered	Reconsidered	Total
Nongreen	0.42	0.19	0.11
Green	0.32	0.08	0.13
TOST p-values (t = 100)	Considered	Reconsidered	Total
Nongreen	**0.03**	**0.01**	**0.00**
Green	**0.02**	**0.00**	**0.00**

Figure 7.9: Subjects in the leading group did not have their averages decreased

7.4 MTurk study

The results of the peer pressuring study of the previous section were certainly promising, but because they were based on a participant pool of 20 users we felt we needed to gather more evidence for our claims. Thus we we once again turned to the readily available participant pool of Amazon Mechanical Turk. However we were immediately faced with a challenge—how can we construct a crowdsourced task that will be interactive enough and allow us to include a subject-controllable delay? Since we could not present users with the familiar phone interface that they could interact with in any reasonable way, we instead chose to create a game of snake with the same peer pressuring interface, and controllable delay settings to test on participants.

Snake game

To build our user study, we used an open source implementation[47] of the classic game of snake, which was released under an MIT license and thus allowed us to modify and use it as we saw fit. We used the same Python Flask framework to create the study application, and added the same peer pressuring interface from the PKU study. The main portion

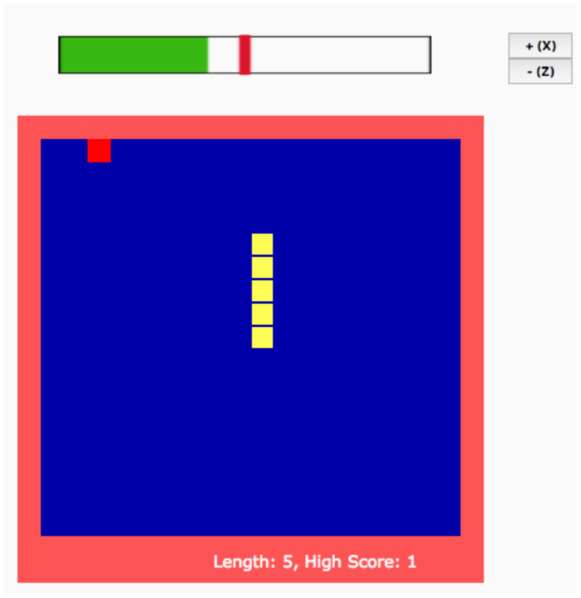


Figure 7.10: The interface visible to the subject during the study, the signaling bar is visible above the game of snake, with controls visible as well.

of the study is shown in Figure 7.10. Since a subject would not have the volume key interface of the PKU study available to them, we added controls for their delay setting via two methods: there were buttons available to the right of the displayed bar, as well as keyboard shortcuts, X for increasing the delay and Z for decreasing the delay. The maximum delay for the game was 250ms, although the interface did not indicate what the delay setting was.

Visibility of the interface

In addition to having two modes for the group delay average, we also wanted to test two variants of displaying the peer pressuring interface. We saw in the original PKU study that reminding a subject of their ability to control the delay (the "Reconsidered" period) increased their delay tolerance envelope, and we additionally did not want to aggravate or confuse the user by giving them 2 simultaneous tasks - playing the game and controlling the delay. We call the two variants the “visible” one, wherein the peer pressuring interface is visible over the whole duration of the study; and the “hidden” variant, where the peer pressuring interface (and associated controls) were only visible between games. In the hidden variant, when a subject was actively playing a game of snake, the interface bar would be made invisible on the webpage, and would only reappear in the period of time between a snake death and subsequent new game start.

Subjects

Our study was IRB approved[2], allowing us to recruit users from the entire US and Chinese crowdsourced participant pools. We advertised the study to participants via the provided framework task discovery mechanism, which allowed potential participants to pick tasks based on description, duration, and compensation. Our selection criteria for each study was that participants had to be geographically located in the United States. We released the study availability in batches of 20, to both ensure that the study server would not become overloaded, and to take advantage of the popularity of newly published tasks. Participant slots were doled out in a “first come first served” fashion, and submissions would be validated manually. If a submission did not pass the validation check, it would be rejected, and that “slot” would be made available to the entire participant pool once again. As part of the study, each participant was also asked to fill out a short demographic survey, and a questionnaire about their familiarity with the game of snake, how long they had been using a smartphone, etc. Each validated participant was rewarded with $0.50 credited directly into their account.

[2]Northwestern IRB Project Number STU00093881

Methodology

The subject would use their own personal computer for the study, and navigate between the MTurk system and EB hosted survey as necessary. All logs would be kept as a database on the EB instance, and verification codes would be submitted and kept in the MTurk management interface. The study would take approximately 9 or 10 minutes to complete.

The subject would begin the survey by following the link provided to them in the HIT description, and be asked to read and agree to the provided IRB consent form. The subject could print out the form, or request a separate copy via email. Since the entire process was carried out online, the users acceptance was recorded in lieu of a signature. After accepting the terms, the subject would be asked to answer a short questionnaire about their demographic information, as well as ranking their familiarity with the game of Snake as well as smartphones in general.

At this point the subject would be provided with instructions on how they would complete the survey. The language of the instructions was identical for all users with one exception: whether or not it contained the environmental prompting. Users would be randomly chosen to be prompted or not, and if so chosen their instruction would contain the following prompt: "We ask that you set the delay as high as you can, while still allowing you to play the game comfortably. Also keep in mind, that as you increase delay, the server this survey is running on will use less energy. In this way, you can reduce your environmental impact by increasing the delay. As you set the delay higher, the snake will take longer to respond to your key presses." If chose to be nongreen, their prompt would be: "We ask that you set the delay as high as you can, while allowing you to play the game comfortably. As you set the delay higher, the snake will take longer to respond to your key presses."

The subject would then begin playing the game. They would be given 3 practice rounds, where no delay was added and no peer pressuring bar was visible. This would give the subject time to familiarize themself with the controls and with the speed of gameplay. Once the 3 practice rounds were over, the peer pressuring interface would appear, as well as a reminder of the controls available. The participant would then play the game for 5 minutes, at which point a button would be made visible for the subject to press to conclude the study. Upon conclusion, the subject would be taken to an exit questionnaire, which asked how environmentally conscious the subject generally considered themselves to be, and, if they had had the visible variant of the interface, asked if they had felt annoyed by its constant presence or found it difficult to use.

7.5 MTurk results

The study involved 400 subjects, 399 of whom we were able to collect meaningful data from, each of whom had played the game for at least 5 minutes. In total there were 4407 games played, with 2109 games having been played by green participants and 2298 games having been played by nongreen participants, and 10,718 total delay changes. The demographics of the study subjects were:

- 98 were enrolled in a College or University, 138 were not.
- 196 were male, 203 were female.
- 190 were chosen to be green, 209 were chose to be nongreen.

Green	Interface	Group	Count
No	Visible	Low	55
No	Visible	High	60
No	Hidden	Low	52
No	Hidden	Hidden	42
Yes	Visible	Low	62
Yes	Visible	High	43
Yes	Hidden	Low	38
Yes	Hidden	High	47

Figure 7.11: Division of number of study subjects among the groups.

- 91 were between the ages of 18 to 25, 158 were between 26 and 35, 91 were between 36 and 45, 41 were between 46 and 55, and 18 were 56 or older.

The counts of users as divided up by our 3 random variables (green, group, and interface) are presented in Figure 7.11. All of the numerical results presented are in terms of percentage of the maximum delay, for example 0% would indicate 0ms of delay, 50% would indicate 125ms, and 100% would indicate 250ms.

Area under the curve

The average values and standard deviations for the envelope of subjects are presented in Figure 7.12. Surprisingly, we find that once again there is no difference between the green and nongreen groups, despite the introduction of a peer pressuring element! We do find that the trailing group had a higher average delay setting than the leading group. Our interpretation of these results is that the lack of a mobile interface has put users back in the mindset of their expectations of performance with a non-mobile device, namely, the personal computer they complete the user study on. However, given that we do still see differences between the leading and trailing groups, we wanted to investigate this difference further.

	Nongreen	Green	Leading	Trailing	Visible	Hidden
Avg	57.14	58.59	55.71	60.12	58.43	57.34
StdDev	17.87	19.20	17.10	19.71	19.43	17.75
T-Test	0.44		**0.02**		0.56	
TOST (t = 5)	**0.03**		0.38		**0.02**	

Figure 7.12: Average value and standard deviation of the user delay envelope, as calculated via the Area Under the Curve Method.

	Leading		Trailing	
	Nongreen	Green	Nongreen	Green
Avg	54.45	57.04	59.93	60.34
StdDev	14.90	19.06	20.14	19.20
T-Test	0.28		0.95	
TOST (t = 5)	0.16		0.06	
	Nongreen		Green	
	Low	High	Low	High
Avg	54.45	59.93	57.04	60.34
StdDev	14.90	20.14	19.06	19.20
T-Test	**0.03**		0.24	
TOST (t = 5)	0.27		**0.02**	

Figure 7.13: Average value and statistical comparisons when splitting along green/nongreen and leading/trailing reveals no hidden effects.

In order to make sure that we were not missing out any effects being hidden by combining too many groups together, we also decompose the data amongst the 4 groups created by splitting on high/low and green/nongreen, and present the results in Figure 7.13. At this decomposition level we find no clear trends of differences between any of the groups, which suggests that we may be seeing two effects—environmental prompting and peer pressuring—colliding in this study, and yielding little effect in total.

Maximum by interval and last Level

The results of analyzing the data via the maximum by interval and last level methods largely support the area under the curve findings, and are presented in Figures 7.14

Interval	Test	Green vs Nongreen	High vs Low	Visible vs Hidden
	T-Test	0.16	**0.02**	0.67
0	TOST	0.37	0.72	0.10
	T-Test	0.61	**0.00**	0.10
5	TOST	**0.04**	0.84	0.27
	T-Test	0.87	**0.01**	0.17
10	TOST	**0.02**	0.65	0.20
	T-Test	0.73	**0.00**	0.21
20	TOST	**0.04**	0.80	0.19
	T-Test	0.26	**0.00**	0.69
30	TOST	0.20	0.89	0.06
	T-Test	0.07	**0.00**	0.66
60	TOST	0.49	0.87	0.09
	T-Test	0.13	**0.01**	0.71
120	TOST	0.41	0.77	0.09

Figure 7.14: No clear trends emerge for any interval

	Nongreen	Green	Leading	Trailing	Visible	Hidden
Avg	57.43	56.69	54.47	59.90	56.46	57.58
StdDev	30.44	29.99	28.74	31.52	30.11	30.32
T-Test	0.81		0.07		0.72	
TOST (t = 5)	0.08		0.56		0.10	

Figure 7.15: Average value and standard deviation of the user delay envelope, as calculated via the Area Under the Curve Method.

and 7.15. The last level method only gives us 93% confidence that the leading and trailing groups behave the same, but this remains close to our results so far.

Environmentally conscious users do not set higher delays

In addition to asking the usual set of demographic questions, we also asked the subjects of this study an additional question—to rank on a scale of 1 to 5 how environmentally conscious they deemed themselves to be "with 1 meaning not environmentally conscious and 5 meaning very environmentally conscious". In Figure 7.16 I enumerate the average delays of all users split by the self ranking, as well as the delays for both the green and

Env conscious?	1	2	3	4	5
Count	29	48	120	144	58
Avg delay	57.66	63.35	61.26	59.55	58.38
Green delay	52.68	62.89	60.40	58.32	55.80
Nongreen delay	56.35	59.77	58.58	55.46	56.47

Figure 7.16: Average delay setting does not increase with self-reported environmental consciousness.

nongreen population. We would expect the average delay setting for more environmentally conscious users to increase in a somewhat linear fashion, but instead see that it peaks at a ranking of 2.

7.6 Conclusions

The user studies of this chapter provide us with yet more surprising results—a small user study of 20 participants demonstrates strong and clear evidence for the efficacy of the peer pressuring mechanism, but a much larger study of 400 users comes up with no difference between the subjects that were environmentally prompted and those that were not. As mentioned in Section 7.5, it is our belief that the lack of a mobile interface are likely what caused the negative results of the MTurk study. However, the strong results of the second portion of the Peking University study motivates further study of the peer pressuring effect, with larger scale and mobile-based setups.

Chapter 8

How can we best communicate with the user?

In this chapter I discuss potential ideas for interfaces that allow meaningful environmental information to be communicated to the user, dynamic envelope information to be gathered from the user, and for virtue signaling to be present in a user's interactions. A few rudimentary interfaces have been used already in the studies that have been undertaken over the course of my doctoral work, and I will discuss these as well, along with their strengths and weaknesses.

8.1 Existing interfaces

Two basic peer pressuring interfaces have been used in our user studies thus far - the simple presentation of current delay settings used in the initial portion of the PKU study, and the group delay comparison used in the latter half of the PKU study as well as the subsequent Mechanical Turk study. These interfaces have been good for evaluating the initial opportunities of both the size of the user delay tolerance envelope and reaction to peer pressuring, however they have many shortcomings in the larger context of this thesis work.

For one thing, there is no relaying of dynamic environmental friendliness information back to the end user, one of my hopes was to able to provide dynamic information back to the user as to both how green they were currently being, as well as how their actions over time were contributing to the datacenter. The current interfaces use the delay setting as a sort of instantaneous measure of greenness, but this falls short of my ultimate vision. Additionally, this interface does not actually signal any information about the user to the outside world, it simply purports to to study the psychological effects of this on the user.

We have also used a rudimentary mechanism for deriving user satisfaction in our various study—self reported satisfaction. In cases where we sought to measure it, we would prompt the user to rate their satisfaction on a Likert scale, assessing how they felt at that moment. One of the tradeoffs of such an approach is that we are limited in how many assessments we can get over time—the more often a user is prompted to rank their satisfaction, the more distracted they become from the task at hand, and the more any dissatisfaction is likely to be due to the prompting itself rather than from any effects that are being measured.

8.2 Gathering information from users

One avenue of work on the interface that could prove highly valuable would be exploring new mechanisms for assessing the satisfaction of the end user. Related work [73, 101, 116, 108] has shown that by using the signals from biological feedback sensors, user annoyance events can be predicted and monitored. By using such sensors, we might be able to directly measure the satisfaction of a user as they are using a mobile application, and be able to adjust their delay settings accordingly. Work has been done by Peter Dinda, my adviser, in creating a mobile bio-sensor platform that could be used to just such an end.

8.3 Group dynamics

We have seen the effects of placing users in the trailing group of user tests, that is, putting them in a situation where they perceive their performance to be lesser than that of their surrounding group. It is important to note that at no time during any study were subjects told that they were explicitly being compared to the group, or that if they consistently did worse than the group that their outcome would be negative in any way, and yet we still saw that the effects came through. This should be hardly surprising, as the competitive nature of humans is well known, and there are in fact entire workshops devoted to studying such gamification effects [24].

One interesting question that I would like to answer is this: what is the most optimal group unit for effecting behavioral changes? By group unit I mean here what boundaries (if any) should be imposed upon the group that a users greenness is being compared to? Some initial proposals for testing include:

- Based on social network connections: we could "piggy-back" off the groups that users already create on a social networking site, such as Facebook, or create a social networking feature within the signaling interface. In either case, the idea is that social connections already contain inherent meaning, and a user is likely to care more about how they compare with their social connections, thus potentially boosting the effects. Fitbit has introduced group-based activity tracking and leaderboards, and has produced positive results [137, 77].

- Based on geographic proximity: since the interface would be running on a mobile device that is already capable of geolocation, the comparison could be made by grouping all users within a certain geographic radius. This could mitigate potential problems such as time-zone mismatches that could exist in the social network option[1]. This approach could also utilize some of the outward signaling mechanisms of the following section.

- Based on national/regional groups: One of the potential downfalls of the earlier two methods is that they may produce groups that are simply too small to provide meaningful comparisons. By comparing a user against other national or regional (for example, state) users, we are assured of having a large enough population pool, and also provide a comparison against users who have a good probability of being in similar situations and environments.

8.4 Outward signaling

The methods discussed thus far in this Chapter provide a means of comparing a users' greenness against people who are participating in the system/interface, but what if we could also include signals that work in general? For example, part of the biosensor platform mentioned earlier is an LED that is affixed to the platform in such a way as to be visible to anyone. Introducing a method where it would be immediately obvious to on-

[1]If your friend is halfway across the world, why would you care how green they are being? They are probably asleep!

lookers how green a person was being might also influence their behavior. Of course, the tradeoff to avoid there would be not making the externally facing signal so obnoxious as to dissuade the user from wearing it. Some initial thoughts for signals to produce would be:

- Brightness of light: a more brightly lit LED would indicate a higher greenness. This would avoid any “punishing” effects, as lower green scores would simply not pick up the attention of others. Depending on the range of values, it may be difficult to meaningfully compare between people who have similar greenness.
- Pulse pattern: better scores could produce slow, soothing pulsing patterns, whereas worse scores could produce more strobing effects. The potential downside would be having the negative signal be too annoying or too distracting. We also do not want to explicitly punish users too much, if at all, for fear of user abandonment.
- Activation of signal: should the signal be visible at all times? Or should there be “activation” events, such as when a person begins to move, when there are more likely to be other people around who would see this signal?

Chapter 9

Related work

In order to put my thesis into context, I now give an overview of related work. I begin by giving an overview of datacenter growth and energy consumption. Next, I discuss various methods of attaining efficient datacenters. I then discuss datacenter workload characteristics. Following this, I motivate using user satisfaction with previous works. Finally, I discuss existing attempts at informing users of their impacts on sustainability, and discuss related work in studying the effect of delays on user satisfaction.

9.1 Datacenters

As mentioned in Section 1, as application complexity and user-base population have increased, so too have the datacenters backing those applications grown. By late 2011, Google's datacenter operation was drawing 260 megawatts of continuous power, with each individual Google search estimated to "cost" 0.3 watt-hours of energy [49], and the power consumption of their combined datacenters jumped from 260MW in 2011 to an estimated 3.2GW in 2015 [36, 72]. In 2009, a single Amazon datacenter was operating at 15 megawatts [57].

In 2007, the Environmental Protection Agency estimated that datacenters had consumed 61 terawatt-hours of energy in the previous year, taking up 1.5% of the total U.S. consumption that year. They also noted that datacenters were the fastest growing consumers of electricity, and predicted that by 2011 an additional 10 power plants would have to be built to be able to satisfy the growing demand of datacenter energy consumption [127].

One of the ever-rising concerns associated with datacenters is that they are not energy efficient. Research studies going as far back as the 90s (e.g. [96]), as well as more modern studies such as [11] estimate that most datacenters are only 10–30% utilized. More recent surveys from the NRDC claim that utilization is only at 12–18% [28]. Some have even claimed that datacenters, especially as they continue to grow, are environmentally hazardous and unsustainable [48].

More recent work estimates that cloud datacenters consume more than 2.4% of global electricity [93]. Perhaps more worryingly, this consumption is expected to grow at a rate of 15-20% annually [92]. In terms of the environmental impact, in 2011 it was estimated that datacenters produced CO_2 emissions equal to 2% of total global CO_2 emissions [17]. A further report from 2016 estimates that datacenter energy consumption jumped from 1.4% to 1.8% from 2010 to 2014 [113].

9.2 Datacenter sustainability

As both power and user demands on datacenters have risen, much work has been done to make datacenter operations more energy efficient. Generally, the work is split into two high-level considerations: attempting to keep the lowest number of servers powered on at any given time, and attempting to optimize the computational efficiency of servers that are currently running.

The traditional approach to datacenter provisioning (which is still widely used in the industry) is to estimate the peak request rate, and to keep on the amount of servers on [22, 62, 129], often overestimating the peak request rate by a factor of 2 [74].

In the server provisioning vein, AutoScale [44] is arguably the state-of-the-art. In contrast to prior work, which attempts to power servers up to provide enough capacity to satisfy an unknown *future* workload, AutoScale instead conservatively powers servers down once there is “slack” in the system, and powering servers up as incoming requests dictate.

Other examples of adapting, in an energy efficient manner, to the offered workload include dynamic voltage and frequency scaling (DVFS) for servers [31], coordinated decisions across the data center [97, 43], and consolidation within the datacenter [128].

Recently the notion of trying to make datacenters more aware of both the availability of green energy sources [30] as well as datacenter applications that are aware of their carbon footprint [29] have been explored by the group of Stewart et al.

With the recent craze of applying various Deep Learning techniques to every field, there has of course also been work in applying machine learning methods to saving energy at the datacenter. Google has applied their recently acquired DeepMind division to predict temperatures in their datacenters and adaptively adjust their cooling systems, reducing cooling costs by up to 40% [25]. Work done by Memik, Zheng, et al at Northwestern University has achieved reductions in power consumption of 5% in the average, and 17% in the optimal case by using machine learning methods to inform task placement on nodes.

9.3 Datacenter traffic

Work done in the 90's to characterize internet traffic characteristics (such as [9, 5]) and to create traffic generators such as SURGE [10] focused on the prevalent traffic on the internet of the time—requests for single websites and their linked (or local) contents. A survey of historical datacenter traces can be found in the AutoScale paper [44], and illustrates the diversity of workloads that datacenters are faced with, and must be ready to deal with.

More recent analyses of datacenter traffic characteristics, such as word by Ersoz et al [40], which used simulated internet auction site traffic created using RUBiS [20], have found statistical characteristics that arise from simulated single-usage on a datacenter. Various benchmarks have been designed that that create application workloads on datacenters running internet services [135] or search engines [45].

Benson et al presented the bursty nature of datacenter traffic by examining a number of private and educational clouds [12]. However I am not aware of any work that has attempted to either characterize or generate traffic created by mobile devices such as smartphone.

9.4 Informing users of energy

The common thread of these sustainability approaches has been that they are all focused on reducing energy exclusively at the datacenter side, and must simply cope with any choices made by end users that affect offered workloads. Most attempts at bringing the user into energy efficient choices have been focused in smart home and heating/cooling scenarios, with initial surveys suggesting that once users are made aware of their energy repercussions, they are willing to play a more active role in reducing their energy footprint [50], especially when exposed to interfaces that offer them feedback and personalization [1]. However most systems today, such as the Nest thermostat [98] still attempt to make the decisions for the user, instead of asking the user how satisfied they are.

9.5 Considering user satisfaction

Datacenter systems are concerned with meeting a contractually agreed upon SLA, which is an analogue for service times that satisfy end user satisfaction. However this user satisfaction is never directly asked, and instead is inferred from the success of a service, as compared to competitors. Work done in the Empathic Systems Project aims to empower the user by including their satisfaction feedback as a signal for resource allocation and provisioning. The overall goals of the project are outlined in [34], and many works from the project have shown that including the user satisfaction is beneficial [123, 117].

Lately there has been a recognition of datacenter-based metrics such a SLA or *quality of service* (QOS) approaches may not capture enough information about the factors that influence *user-perceived satisfaction*, that is, how the performance of the service or application is experienced by individual users. One of the responses has been to attempt to quantify the *quality of experience*, or QoE, which is simply the overall acceptability of an application or service as perceived subjectively by an end-user [35]. Much of the work done in this area has been focused on finding methods for judging QoE in video streaming services [111, 134, 100].

Much like the related work of the previous section, recent work such as that by Shorfuzzaman [114] still does not attempt to ask end-users how satisfied they are, but rather consider "User satisfaction rate [to be] the percentage of users whose QoS requirements are met." As I have shown in my doctoral work, user satisfaction under the same conditions is highly variable, and approaches such as these miss out on the opportunity to tap into more tolerant users.

9.6 Traffic shaping

Traffic shaping has had its greatest success in computer networks. It originated in ATM networks [70, 104] and then expanded widely [46, 39], for example into DiffServe [14], and today is widely deployed. The user-centric traffic shaping concept is named by analogy, but an important distinction is that we focus on *shaping the users' offered workload to the cloud*, as well as *shaping the users' perception of the performance that workload receives.*

Google uses traffic shaping inside of their datacenters, and has patented methods by which user requests can be redirected based on application layer information as to the potential tasks and sub-tasks of a request [19], as well as by selectively adding delays if responding to the request is deemed to not be immediately critical and current resource contention is high [121].

Akhshabi et al have shown that by enabling traffic shaping on servers which serve streaming videos, the amount of "oscillation" of quality of streams caused by competing clients can be drastically reduced [4]. Getting close to the end-user side of things, traffic shaping methods for increased HTTP stream have been proposed by Villa et al [130] on a geographically regional grouping, and by Houdaille and Gouache [63] at the level of home gateways.

As far as I am aware, there have been no attempts to shape the traffic coming directly from end-users for the purpose of energy cost reduction.

9.7 Satisfaction and delay

A question that has not been researched widely is this: what does it take for a mobile application user to abandon an application? Based on information gathered from news reports, Facebook recently began introducing a variety of bugs and interface issues, causing lags and crashes on their native Android Application [38]. According to the article, no matter how unstable the app was made, the user loss rate was effectively zero.[1] Of course, it is important to note that Facebook is an enormously successful and popular social network, and operates with little competition so some users may feel like they have no option but to stay with the application despite any issues, but this still presents some indication that mobile application users may not be as sensitive to delay as previously thought.

9.8 Green interfaces

There has been a lot of work done in the HCI community exploring methods of effecting behavioral changes in the real of environmental activities. Work by Dillahunt et al has shown that feedback can be a powerful motivator of behavioral change – they created a survey in which a users actions affected a "virtual polar bear" [32]. They found that seeing an immediate effect on this virtual animal increased environmentally responsible behavior, especially when users became emotionally attached. Similar behaviors were found based around an early 90's children's toy, dubbed the "Tamagotchi Effect" [61], wherein emotional attachments to a virtual being caused changes in user behavior.

[1] I reached out to the author of the article for more information and data, but was unable to gain any more insights.

Work by Froehlich and his group used this study to their advantage, and they created the MyExperience interface [41] which gave users feedback with interactive backgrounds such as a blossoming apple tree or a polar ice cap that grew and fostered more polar bears and other arctic critters.

Froehlich's group used the MyExperience interface to conduct a couple of studies, including Ubigreen [42], wherein users were encouraged to take green methods of transportation such a public transit, biking, or walking; and Ubifit [26], where users were encouraged to partake in a more active lifestyle, doing cardiovascular, resistance, or other training. In both studies, users found that the reward of seeing their effects in the interface motivated them to increase the behaviors that led to positive feedback.

As far as I am aware, there has not been any work done in attempting to inform users of the effects of their behavior on the datacenter, which gives me an opportunity for tapping into the existing desire to be more environmentally friendly to be able to change a user's traffic.

Chapter 10

Conclusions

In cloud-backed mobile applications, there are always tradeoffs between delivering content to the end-user. Based on studies by large cloud providers such as Google and Amazon, the sensitivity to latency of the end-user has been estimated as very high, which has meant that most, if not all, work on reducing the energy footprint of cloud datacenters has been on the datacenter side. I claim that in the mobile space, this sensitivity is actually much lower, and can in fact be further mitigated by contextualizing users in an environmental frame of mind. Furthermore, these effects increase when a signaling mechanism is introduced, which allows a users greenness to be broadcast to the world and compared to others in their surroundings.

Many of these findings are surprising and contraindicate previously mentioned findings, so it is important to remind oneself that previous studies were carried out in a non-mobile environment, and are not capturing the same kind of user experience and expectations that the studies I have conducted with my collaborator.

Along with the summary of work, this final chapter lists the major contributions of my doctoral work, as well as avenues for future work based on the work already completed.

10.1 Contributions

The major contributions of my dissertation are:

- **Establishing that modern systems are in fact sometimes running too fast, and that by selectively delaying parts of user-interface code, we create an opportunity for energy savings while not irritating the end user.** I show this by delaying end-user JavaScript with simulated sleep calls, and create the JSSlow proxy to inject these calls into the JavaScript of popular websites. Most power savings are realized from advertising and buggy code, with an average of 5% reduction in energy found during a user study.

- **Establishing the opportunity for using delays in cloud-backed mobile applications by demonstrating that user delay tolerance is higher than previously indicated.** By conducting studies in which delays were randomly injected into real world, popular applications, and also by allowing users to set their own delays, I have shown that delays of up to 750ms are acceptable for users.

- **Showing that traffic can be shaped while staying within the demonstrated user delay tolerance.** I create and simulate an initial algorithm for shaping user traffic based on traces collected during user studies, and show that the traffic can be made to more closely resemble desired characteristics, while not introducing delays larger than those showed to be acceptable.

- **Showing that only equating larger delays with environmental friendliness is not enough to cause behavioral changes.** We initially assumed that by telling users that by allowing the performance of their mobile applications degrade they would allowing the backing cloud to operate in a more environmentally friendly setting, they would be willing to tolerate additional delays. We show that this is in fact not the case, at least for the demographics of the studies we carry out.

- **Showing that peer pressure mechanisms can create power for environmental prompting.** When we added a signaling interface to the smartphone user study we saw large and obvious effects added to the environmental prompting. In the Mechanical Turk study carried out we saw effects of the group comparison, but ultimately failed to produce an environmental differentiation. This result strengthens our assertion that users have an altogether different perception and experience in mobile environments, and that we have a unique opportunity to effect changes in that arena.

10.2 Future work

I now lay out some potential avenues for future work. Having laid the groundwork for establishing the size of the user delay tolerance envelope, there is much work left to be done in expanding the envelope with signaling interfaces, and in building a system that would take individual envelopes into account and apply traffic shaping as appropriate.

Design a real world signaling mechanism. We tested a simple peer pressure mechanism in our user studies - showing users a (simulated) group average and how they compared to it. This approach produced positive results, but we think that more complex interfaces would be able to provide an even bigger effect. We envision a system where a user would be broadcasting a literal signal to the world around them, such as a colored LED that would indicate just how environmentally friendly they were being.

Study more environmentally conscious populations. We ran our user studies on populations located in the Unites States and China, and found that just environmental prompting was not enough to effect behavioral changes. It could prove interesting and perhaps useful to repeat these studies on populations that are known to be more environmentally friendly, in countries where attention to recycling and automotive emissions have already produced changes in the general population, such as Germany or Austria [3], or Sweden—where the country produces so little trash that they have begun importing it from other countries for energy production [15]. If studies in such countries demonstrated more power of the prompting, it could be a good indication of such effects spreading to the US as well in the future[1].

Deploy traffic shaping at an individual user level and study the results in the aggregate. We created one algorithm for shaping user traffic, and analyzed it by running it on traces collected from earlier user studies. It would be good to develop additional shaping algorithms, different shaping targets, and deploy a system which puts them into effect on many users, studying what the effects on both their traffic patterns and behavior they would have.

Design and study mechanisms for deducing user satisfaction dynamically. Currently we do not attempt to infer a users satisfaction in any interfaces, and rather let the user control their delay settings manually. By incorporating additional feedback, such as from sensors that could be worn by the user, or other feedback mechanisms such as shaking or tapping a phone, we could design an interface which learns when a user is dissatisfied,

[1]Though hopefully the idea that mainstream US culture adopts for environmentally friendly habits is not just the optimistic thinking of yours truly.

and attempts to modulate their delay setting accordingly.

Design a dynamic representation of the environmental friendliness of a user. We currently use the default delay setting as a proxy for how environmentally friendly a user is at any given point in time, but this will surely prove to be too simplistic of a measure in the long run. We have seem from related works that interfaces which track behavior over time, and provide emotional feedback, such as the virtual polar bear or the Tamagotchi, can encourage behavioral changes in users. We would like to develop a system and interface which leverages this power in encouraging users to change their behavior in environmentally positive ways, while not overly discouraging them for short term behaviors.

Context of performance expectations An open question in why the delay tolerance envelope of users has been found to be so high in these studies, is how much of this is due to users having low expectations of performance in a mobile environment due to poor quality cellular data networks, or slower hardware / unoptimized mobile software? As both hardware and cellular networks continue to make make progress, it would be very interesting to repeat our studies, especially our "in-the-wild" study done on real applications, as well as the lab study at PKU in which users controlled their delay setting, and seeing if the results begin to change at all.

Quick hardware idle states One of the reasons for the massive overprovisioning of server nodes in datacenters is the very large amount of time (many minutes) of powering on those nodes. As progress is made in lower power states in server hardware this power on time may become less problematic, meaning that dealing with burstiness may become easier. This may not entirely reduce the need for the shaping methods I have proposed in this work, but rather shift the focus onto what shaping targets would be optimal, and what kind of characteristics would then be most beneficial for datacenter operators.

References

[1] Wokje Abrahamse, Linda Steg, Charles Vlek, and Talib Rothengatter. A review of intervention studies aimed at household energy conservation. *Journal of environmental psychology*, 25(3):273–291, 2005.

[2] Advanced Scientific Computing Advisory Committee. The opportunities and challenges of exascale computing. Technical report, Department of Energy, Fall 2010.

[3] European Environment Agency. Highest recycling rates in germany and austria – but uk and ireland show fastest increase, 2013. [Online; accessed 2016-8-11].

[4] Saamer Akhshabi, Lakshmi Anantakrishnan, Constantine Dovrolis, and Ali C. Begen. Server-based traffic shaping for stabilizing oscillating adaptive streaming players. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '13, pages 19–24, New York, NY, USA, 2013. ACM.

[5] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana De Oliveira. Characterizing reference locality in the www. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 92–103. IEEE, 1996.

[6] Amazon.com. AWS Elastic Beanstalk, 2011–. [Online; accessed 2016-08-15].

[7] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, 2005.

[8] Andrew Appel. *Compiling with continuations*. Cambridge University Press, Cambridge New York, 1992.

[9] Martin F Arlitt and Carey L Williamson. Web server workload characterization: The search for invariants. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 126–137. ACM, 1996.

[10] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 151–160. ACM, 1998.

[11] L. Barroso and U. Hoelzle. The case for energy-proportionate computing. *IEEE Computer*, 40(12):33–37, December 2007.

[12] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[14] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, Network Working Group, December 1998.

[15] Elisabeth Braw. Dirty power: Sweden wants your garbage for energy, 2015. [Online; accessed 2016-8-10].

[16] P.G. Bridges, D. Arnold, and K. Pedretti. Vm-based slack emulation of large-scale systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2011.

[17] Richard Brown et al. Report to congress on server and data center energy efficiency: Public law 109-431. *Lawrence Berkeley National Laboratory*, 2008.

[18] Michael Buhrmester, Tracy Kwang, and Samuel D Gosling. Amazon's mechanical turk a new source of inexpensive, yet high-quality, data? *Perspectives on psychological science*, 6(1):3–5, 2011.

[19] F.J. Castaneda, J.K. Horvath, and A.W. Wrobel. Application layer synchronous traffic shaping, August 14 2012. US Patent 8,243,597.

[20] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *ACM Sigplan Notices*, volume 37, pages 246–261. ACM, 2002.

[21] Pew Research Center. Pew research center smartphone ownership report 2013. Technical report, Pew Research Center, http://www.pewinternet.org/2013/06/05/smartphone-ownership-2013/, 2013.

[22] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.

[23] S. Cheshire. Latency and the quest for interactivity. In *White paper commissioned by Volpe Welty Asset Management, LLC, for the Synchronous Person-to-Person Interactive Computing Environments Meeting*, 1996.

[24] CHI Italy. *Proc. Workshop on Ubiquitous games and gamification for promoting behavior change and wellbeing (CHI Italy, Trento, Italy, September 2013)*, Trento, Italy, 2013.

[25] Jack Clark. Google cuts its giant power bill with deepmind-powered ai, 2016. [Online; accessed 2016-8-25].

[26] Sunny Consolvo, David W McDonald, Tammy Toscos, Mike Y Chen, Jon Froehlich, Beverly Harrison, Predrag Klasnja, Anthony LaMarca, Louis LeGrand, Ryan Libby, et al. Activity sensing in the wild: a field trial of ubifit garden. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1797–1806. ACM, 2008.

[27] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[28] Pierre Delforge. Data center efficiency assessment, August 2014.

[29] Nan Deng, Christopher Stewart, Daniel Gmach, and Martin Arlitt. Policy and mechanism for carbon-aware cloud applications. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 590–594. IEEE, 2012.

[30] Nan Deng, Christopher Stewart, Daniel Gmach, Martin Arlitt, and Jaimie Kelley. Adaptive green hosting. In *Proceedings of the 9th international conference on Autonomic computing*, pages 135–144. ACM, 2012.

[31] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini. Coscale: Coordinating cpu and memory systems dvfs in server systems. In *Proceedings of the 45th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO 2012)*, August 2012.

[32] Tawanna Dillahunt, Geof Becker, Jennifer Mankoff, and Robert Kraut. Motivating environmentally sustainable behavior changes with a virtual polar bear. In *Pervasive 2008 Workshop Proceedings*, volume 8, pages 58–62, 2008.

[33] DIMPLE.IO. Dimple.io nfc android buttons, 2016. [Online; accessed 2016-8-24].

[34] Peter Dinda et al. The user in experimental computer systems research. In *Proceedings of the Workshop on Experimental Computer Science*, June 2007.

[35] C Dvorak. Definition of quality of experience (qoe). Technical report, ITU - International Telecommunication Union, Reference: TD 109rev2 (PLEN/12), 2007.

[36] Brian Eckhouse. Google buys 781 megawatts of wind, solar power in three nations, 2015. [Online; accessed 2016-8-22].

[37] S. Ecma. Ecma-262 ecmascript language specification, 2009.

[38] Amir Efrati. Facebook's android contingency planning. Technical report, The Information, https://www.theinformation.com/facebooks-android-contingency-planning, 2016.

[39] A. Elwalid and D. Mitra. Traffic shaping at a network node: Theory, optimal design, and admission control. In *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 97)*, April 1997.

[40] Deniz Ersoz, Mazin S Yousif, and Chita R Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 59–59. IEEE, 2007.

[41] Jon Froehlich, Mike Y Chen, Sunny Consolvo, Beverly Harrison, and James A Landay. Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 57–70. ACM, 2007.

[42] Jon Froehlich, Tawanna Dillahunt, Predrag Klasnja, Jennifer Mankoff, Sunny Consolvo, Beverly Harrison, and James A Landay. Ubigreen: investigating a mobile tool for tracking and supporting green transportation habits. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1043–1052. ACM, 2009.

[43] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2009)*, June 2009.

[44] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Mike Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4):1–26, November 2012.

[45] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shiming Gong, et al. Bigdatabench: a big data benchmark suite from web search engines. *arXiv preprint arXiv:1307.0320*, 2013.

[46] Leonidas Georgiadis, Roch Guerin, Vinod Peris, and Kumar Sivarajan. Efficient network qos provisioning based n per-node traffic shaping. *IEEE/ACM Transactions on Networking*, 4(4):482–501, 1996.

[47] Patrick Gillespie. Javascript snake, 2015. [Online; accessed 2016-8-15].

[48] J. Glanz. The cloud factories: Power pollution and the internet. New York Times, September 22, 2012.

[49] J. Glanz. Google details, and defends, its use of electricity. New York Times, September 8, 2011.

[50] David L Goldblatt, Christoph Hartmann, and Gregor Dürrenberger. Combining interviewing and modeling for end-user energy conservation. *Energy Policy*, 33(2):257–271, 2005.

[51] Google. Android webview. `http://developer.android.com/reference/android/webkit/WebView.html`.

[52] Google. Closure compiler. `https://developers.google.com/closure/compiler/`.

[53] Speed matters: Google research blog. `http://googleresearch.blogspot.com/2009/06/speed-matters.html`. Accessed: 2016-05-21.

[54] Ashish Gupta, Bin Lin, and Peter A. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, June 2004.

[55] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. Diecast: testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08. USENIX Association, 2008.

[56] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76. IEEE, 2016.

[57] James Hamilton. Where does the power go in high scale data centers? Keynote of ACM SIGMETRICS 2009.

[58] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04. IEEE Computer Society, 2004.

[59] Hecorat. Az screen recorder (version 4.1.0) [mobile application software]. Technical report, Hecorat, https://play.google.com/store/apps/details?id=com.hecorat.screenrecorder.free, 2016.

[60] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93. ACM, 1993.

[61] Andreas Holzinger and Hermann Maurer. Incidental learning, motivation and the tamagotchi effect: Vr-friends, chances for new ways of learning with computers. *Computer Assisted Learning (CAL)*, 99, 1999.

[62] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 270–279. ACM, 2008.

[63] Rémi Houdaille and Stéphane Gouache. Shaping http adaptive streams for a better user experience. In *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, pages 1–9, New York, NY, USA, 2012. ACM.

[64] Gang Huang, Huaqian Cai, Maciej Swiech, Xuanzhe Liu, and Peter Dinda. Delaydroid: An instrumented approach to reducing tail-time energy of android apps. *Science China: Information Sciences*, 2016.

[65] Apple Inc. Apple updates ios to 6.1. Technical report, Apple, http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html, 2013.

[66] Intel. Intel architecture instruction set extensions programming reference. `http://software.intel.com/sites/default/files/m/3/2/1/0/b/41417-319433-012.pdf`, 2012.

[67] Intel. Intel software development emulator, November 2012. v. 5.31.0.

[68] Intel. Intel 64 and ia-32 architectures software developer's manual volume 3c, chapter 32. `http://download.intel.com/products/processor/manual/325384.pdf`, 2013.

[69] Panagiotis G Ipeirotis. Demographics of mechanical turk. Technical report, NYU, 2010.

[70] Raj Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.

[71] Inc Joyent. Nodejs platform. `http://www.nodejs.org/`.

[72] Peter Judge. The truth is: data center power is out of control, 2016. [Online; accessed 2016-8-23].

[73] A. Kapoor, W. Burleson, and R. W. Picard. Automatic prediction of frustration. *Intl. Journal of Human-Computer Studies*, pages 724–736, August 2007.

[74] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM computer communication review*, 41(1):102–108, 2011.

[75] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.

[76] Alexander Kudryavtsev, Vladimir Koshelev, Boris Pavlovic, and Arutyun Avetisyan. Modern hpc cluster virtualization using kvm and palacios. In *Proceedings of the Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, held in conjunction with ICAC 2012*. ACM, 2012.

[77] Masaaki Kurosu. *Human-Computer Interaction. Novel User Experiences: 18th International Conference, HCI International 2016, Toronto, ON, Canada, July 17-22, 2016. Proceedings*, volume 9733. Springer, 2016.

[78] Jack Lange, Peter Dinda, and Sam Rossoff. Experiences with client-based speculative remote display. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2008.

[79] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[80] John R. Lange, Peter Dinda, Kyle C. Hale, and Lei Xia. An introduction to the palacios virtual machine monitor—version 1.3. Technical Report NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, November 2011.

[81] John R. Lange, J. Scott Miller, and Peter A. Dinda. Emnet: Satisfying the individual user through empathic home networks. In *Proceedings of the 29th IEEE Conference on Computer Communications (INFOCOM)*, March 2010.

[82] John R. Lange, Kevin Pedretti, Peter Dinda, Patrick G. Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11. ACM, 2011.

[83] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[84] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing)*, November 2005.

[85] Bin Lin and Peter Dinda. User-driven scheduling of interactive virtual machines. In *Proceedings of the Fifth International Workshop on Grid Computing*, November 2004.

[86] Bin Lin and Peter Dinda. Towards scheduling virtual machines based on direct user input. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, 2006.

[87] Bin Lin, Arindam Mallik, Peter Dinda, Gokhan Memik, and Robert Dick. User- and process-driven dynamic voltage and frequency scaling. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009.

[88] Leib Litman, Jonathan Robinson, and Cheskie Rosenzweig. The relationship between motivation, monetary compensation, and data quality among us-and india-based workers on mechanical turk. *Behavior research methods*, 47(2):519–528, 2015.

[89] Yabing Liu, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Analyzing facebook privacy settings: user expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 61–70. ACM, 2011.

[90] A. Mallik, J. Cosgrove, R. Dick, G. Memik, and P. Dinda. Picsel: Mearuing user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th International Conference in Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[91] A. Mallik, B. Lin, G. Memik, P. Dinda, and G. Memik. User-driven frequency scaling. *IEEE Computer Architecture Letters*, 5(2), 2006.

[92] Eric Masanet, Arman Shehabi, and Jonathan Koomey. Characteristics of low-carbon data centres. *Nature Climate Change*, 3(7):627–630, 2013.

[93] GI Meijer. Cooling energy-hungry data centers. *Science*, 328(5976):318–319, 2010.

[94] J. Scott Miller, Amit Mondal, Rahul Potharaju, Peter A. Dinda, and Aleksandar Kuzmanovic. Understanding end-user perception of network problems. In *Proceedings of the SIGCOMM Workshop on Measurements Up the STack (W-MUST 2011)*, August 2011. Extended version available as Northwestern University Technical Report NWU-EECS-10-04.

[95] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. Logtm: log-based transactional memory. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, HPCA '06, 2006.

[96] Matt W. Mutka and Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, July 1991.

[97] Ripal Nathuji and Karsten Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, October 2007.

[98] Nest. Nest learning thermostat. Technical report, Nest, http://nest.com/, 2016.

[99] OKCupid. Tamejs javascript framework. `http://www.tamejs.org`.

[100] Kandaraj Piamrat, Cesar Viho, J Bonnin, and Adlen Ksentini. Quality of experience measurements for video streaming over wireless networks. In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*, pages 1184–1189. IEEE, 2009.

[101] R. W. Picard. *Affective Computing*. MIT Press, Cambridge, 1997.

[102] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K.P. Pipe, T.F. Wenisch, and M.M.K. Martin. Computational sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[103] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05. IEEE Computer Society, 2005.

[104] Jennifer Rexford, F. Bonomi, A. Greenberg, and A. Wong. Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches. *IEEE Journal on Selected Areas in Communications*, 15(5):938–950, 1997.

[105] Armin Ronacher. Flask: a micro web framework for Python, 2010–. [Online; accessed 2016-08-15].

[106] Sonia Sachs and Katherine Yelick. Ascr programming challenges for exascale. Technical report, Department of Energy, 2011.

[107] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 183–192. IEEE, 2015.

[108] Matt Schuchhardt, Ben Scholbrock, Utku Pamuksuz, Gokhan Memik, Peter Dinda, and Robert Dick. Understanding the impact of laptop power saving options on user satisfaction using physiological sensors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED 2012)*, July-August 2012.

[109] Donald J 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 biopharmaceutics*, 15(6):657–680, 1987.

[110] Prem Seetharaman and Bryan Pardo. Audealize: Crowdsourcing audio production tools. *Journal of the Audio Engineering Society*, 2016.

[111] René Serral-Gracià, Eduardo Cerqueira, Marilia Curado, Marcelo Yannuzzi, Edmundo Monteiro, and Xavier Masip-Bruin. An overview of quality of experience measurement challenges for video applications in ip networks. In *Wired/Wireless Internet Communications*, pages 252–263. Springer, 2010.

[112] Steven E Sexton and Alison L Sexton. Conspicuous conservation: The prius halo and willingness to pay for environmental bona fides. *Journal of Environmental Economics and Management*, 67(3):303–317, 2014.

[113] Arman Shehabi, Sarah Josephine Smith, Dale A. Sartor, Richard E. Brown, Magnus Herrlin, Jonathan G. Koomey, Eric R. Masanet, Nathaniel Horner, Inês Lima Azevedo, and William Lintner. United states data center energy usage report, 06/2016 2016.

[114] Mohammad Shorfuzzaman. Access-efficient qos-aware data replication to maximize user satisfaction in cloud computing environments. In *2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 13–20. IEEE, 2014.

[115] A. Shye, A. Mallik B. Ozisikyilmaz, G. Memik, P. Dinda, R. Dick, and A. Choudhary. Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, June 2008.

[116] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. Dinda, and R. Dick. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Lake Como, Italy, Nov. 2008 [Nominated for Best Paper Award].

[117] Alex Shye, Yan Pan, Ben Scholbrock, J. Scott Miller, Gokhan Memik, Peter A. Dinda, and Robert P. Dick. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 188–199, Washington, DC, USA, 2008. IEEE Computer Society.

[118] Joao Sousa, Rajesh Balan, Vahe Poladian, David Garlan, and Mahadev Satyanarayanan. Giving users the steering wheel for guiding resource-adaptive systems. Technical Report CMU-CS-05-198, School of Computer Science, Carnegie Mellon University, December 2005.

[119] Srini V. Srinivasan. Lessons learned in building real-time big data systems. In *Proceedings of the 20th International Conference on Management of Data*, COMAD '14, pages 5–6, Mumbai, India, India, 2014. Computer Society of India.

[120] G.L. Steele and G.J. Sussman. Scheme: An interpreter for the extended lambda calculus. *Artificial Intelligence Lab Memo*, 349, 1975.

[121] D.A. Sterling, S. Mathur, and V. Boctor. Traffic shaping based on request resource usage, July 7 2016. US Patent App. 15/053,847.

[122] Maciej Swiech, Huaqian Cai, Peter Dinda, and Gang Huang. Prospects for shaping user-centric mobile application workloads to benefit the cloud. In *Proceedings of the 24th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2016)*, September 2016.

[123] Maciej Swiech and Peter Dinda. Making javascript better by making it even slower. In *Proceedings of the 21st IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)*, August 2013.

[124] Stephen Tarzia, Robert Dick, Peter Dinda, and Gokhan Memik. Sonar-based measurement of user presence and attention. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp 2009)*, September 2009. Poster also appeared in Usenix 2009.

[125] Stephen Tarzia, Peter Dinda, Robert Dick, and Gokhan Memik. Display power management policies in practice. In *Proceedings of the 7th IEEE International Conference on Autonomic Computing (ICAC 2010).*, June 2010.

[126] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 41–50, New York, NY, USA, 2012. ACM.

[127] United States Environmental Protection Agency. Report to congress on server and data center energy efficiency public law 109-431, August 2007.

[128] Akshat Verma and Gargi Dasgupta. Server workload analysis for power minimization using consolidation. In Geoffrey M. Voelker and Alec Wolman, editors, *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[129] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 28–28. USENIX Association, 2009.

[130] Bjørn J Villa and Poul E Heegaard. Group based traffic shaping for adaptive http video streaming by segment duration control. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 830–837. IEEE, 2013.

[131] Anton Vorontsov. Android interactive cpu governor kernel patch. `http://lkml.org/lkml/2012/2/7/483`.

[132] Wandouijia. Wandoujia android application market. Technical report, Wandouijia, https://www.wandoujia.com/, 2016. [Online; accessed 2016-08-22].

[133] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, September 2012.

[134] Bo Wang, Xiangmin Wen, Sun Yong, and Zheng Wei. A new approach measuring users' qoe in the iptv. In *Circuits, Communications and Systems, 2009. PACCS'09. Pacific-Asia Conference on*, pages 453–456. IEEE, 2009.

[135] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: a big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[136] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. In *Proceedings of ACM SIGCOMM '95*, pages 100–113, 1995.

[137] Business Wire. Fitbit introduces "fitbit group health" for corporate wellness, weight management programs, insurers and clinical research, 2016. [Online; accessed 2016-7-23].

[138] Lei Yang, Robert Dick, Gokhan Memik, and Peter Dinda. Happe: Human and application driven frequency scaling for processor power efficiency. *IEEE Transactions on Mobile Computing (TMC)*, 12(8):1546–1557, August 2013. Selected as a Spotlight Paper.

[139] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 233–248. ACM, 2012.

Appendix A

Emulating hardware transactional memory with VMM

In this appendix I describe work that I did with my co-author Kyle Hale, on emulating RTM, Intel's implementation of hardware transactional memory which was added to their Haswell line of processors. The result of this work was published as a full paper in the ROSS workshop in conjunction with HPDC in 2014.

Hardware transactional memory (HTM) [60] is an enduring concept that holds considerable promise for improving the correctness and performance of concurrent programs on hardware multiprocessors. Today's typical server platforms are already small scale NUMA machines. A mid-range server may have as many as 64 hardware threads spread over 4 sockets. Further, it is widely accepted that the growth of single node performance depends on increased concurrency within the node. For example, the U.S. national exascale efforts are crystallizing around a model of billion-way parallelism [2], of which a factor of 1000 or more is anticipated to be within a single node [106]. Given these trends, correct and efficient concurrency within a single node or server is of overarching importance, not just to systems software, but also to libraries and applications.

HTM promises better correctness in concurrent code by replacing locking with transactions over instructions. Unlike locks, such transactions are composable, meaning that it is less likely to introduce deadlock and livelock bugs as a codebase expands. Furthermore, transactions have the potential for running faster than locks because the hardware is able to detect violations of transaction independence alongside of maintaining coherence.

Intel has made HTM a component of its Haswell platform, and chips with the first implementation of this feature are now widely available. This appendix focuses on the restricted transactional memory (RTM) component of Intel's specification. RTM is a bit of a misnomer—it might better be called *explicit* transactional memory. With RTM, the programmer starts, aborts, and completes transactions using new instructions added to the ISA. Our work does not address the other component of Intel's specification, hardware lock elision (HLE), which is a mechanism for promoting some forms of existing lock-based code to transactions automatically—i.e., it is *implicit* transactional memory.

Our appendix focuses on how to extend a virtual machine monitor (VMM) so that it can provide the guest with Intel Haswell RTM capability even if the underlying hardware does not support it. Furthermore, the limitations of this emulated capability can differ from that of the underlying hardware.

There are three primary use cases. The first is in testing RTM code against different hardware models, to attempt to make the code resilient to different and changing hardware. As we describe in more detail in Section A.1, transaction aborts are caused not only by the detection of failures of transaction independence, but also by other events that are strongly dependent on specific hardware configurations and implementations. Hence, a transaction that may succeed on one processor model might abort on another model.

The second use case is to consider potential future RTM implementations, including those that might allow arbitrary length transactions. Current RTM hardware implementations limit transaction length due to cache size and write buffer size limitations. Our system is free of such limitations unless they are explicitly configured. Conceivably, this functionality could also be used to bridge RTM and software transactional memory.

The third use case is in debugging RTM code via a controlled environment. Through emulation, it is possible to introduce abort-generating events at specific points and observe the effects. It is also possible to collect detailed trace data from running code.

We have designed, implemented, and evaluated a system for Intel RTM emulation within the Palacios VMM. Our techniques are not specific to Palacios, and could be implemented in other VMMs as well. Our implementation is available as a part of the open-source Palacios codebase. Our contributions are as follows:

- We have designed a page-flipping technique that allows instruction execution while capturing instruction fetches, and data reads and writes. This technique avoids the need for any instruction emulation or complex instruction decoding other than determining instruction length. This greatly simplifies RTM emulation and could be applied to other services.

- We have designed an emulation technique for RTM based around the page-flipping technique, redo-logging, undefined opcode exceptions, and hypercalls. The technique is extensible, allowing for the inclusion of different hardware models, for example different cache sizes and structures.

- We have implemented the RTM emulation technique in Palacios. The entire technique comprises about 1300 lines of C code.

- We have evaluated our VMM-based RTM emulation technique and compared it with Intel's emulator-based implementation of Haswell RTM in the Software Development Emulator [67]. Our implementation is approximately 60 times faster when a transaction is executing, and has full performance when none are.

Related work: Herlihy and Moss introduced HTM [60]. Recent work by Rajwar, Herlihy, and Lai showed how HTM could be extended such that hardware resource limitations would not be programmer visible [103]. Unbounded transactional memory [7] shows that hardware designs that allow arbitrarily long transactions are feasible, and this work also demonstrated that using such transactions would allow for significant speedups in Java and Linux kernel code. Hammond et al. [58] have argued for using such powerful transactions as the basic unit of parallelism, coherence, and consistency. In contrast to such work, our goal is simply to efficiently emulate a specific commercially available HTM system that will have model-specific hardware resource limitations. By using our system, programmers will be able to test how different hardware limits might affect their programs. However, because the conditions under which our system aborts a transaction are software defined, and the core conflict detection process will work for a transaction of any size, provided sufficient memory is available, our system could also be employed to test models such as the ones described above. IBM has produced their own implementation of HTM in the BlueGene/Q architecture [133].

Our system leverages common software transactional data structures, such as hashes and redo logs. Moore et al. developed an undo log-based hardware transactional memory system [95] which lets all writes go through to memory and rolls them back upon conflict detection. Our emulator rolls a redo log forward on a commit.

A.1 Haswell transactional memory

The Haswell generation of Intel processors include an implementation of hardware transactional memory. The specification for *transactional synchronization extensions* (TSX) has the goals of providing support for new code that explicitly uses transactions, backward compatibility of some such new code to older processors, and allowing for hardware differences and innovation under the ISA-visible model [66]. There are two models supported by TSX, *hardware lock elision* (HLE) and *restricted transactional memory* (RTM). Our focus in this appendix is on RTM.

A.1.1 Restricted transactional memory

In the RTM model, four additional instructions have been added to the ISA: XBEGIN, XEND, XABORT, and XTEST. The system software uses CPUID checks to determine if these instructions are available on the present hardware. If they are executed on hardware which does not support them, a #UD (undefined opcode) exception is generated. Code can use the XTEST instruction to determine if it is executing within a transaction. An RTM transaction is typically written in a form like this:

```
start_label:
     XBEGIN abort_label
     <body of transaction, may use XABORT>
     XEND
success_label:
     <handle transaction commited>
abort_label:
    <handle transaction aborted>
```

The XBEGIN instruction signifies the start of a transaction, and provides the hardware with the address to jump to if the transaction is aborted. The body of the transaction then executes. If no abort conditions arise, the transaction is committed by the XEND.

Conceptually, the core on which the body of the transaction, from XBEGIN to XEND, executes does reads and writes that are independent of those from other cores, and its own writes are not seen by other cores until after the XEND completes successfully. If another core executes a conflicting write or read, breaking the promise of independence, the hardware will abort the transaction, discard all of the completed writes, and jump to the abort label. The code in the body of the transaction may also explicitly abort the transaction using the XABORT instruction. The specific reason is written into RAX so the abort handling code can decide what to do.

Beyond conflicting memory reads and writes on other cores, and the execution of the XABORT instruction, there are numerous reasons why a transaction may abort. These form three categories: instructions, exceptions, and resource limits. Within each category, there are both implementation-independent and implementation-dependent items. One of the benefits of our emulated RTM system is to allow the testing of RTM code under different implementations to bullet-proof it.

Instructions: The XABORT, CPUID, and PAUSE instructions are guaranteed to abort in all implementations. In addition, the specification indicates a very diverse set of other instruction classes may also cause aborts, depending on the specific RTM implementation. Whether or not the following instructions may abort depends on the implementation:

- X87 floating point and MMX instructions

- Instructions that update non-status parts of RFLAGS
- Instructions that update segment, debug, and control registers
- Instructions that cause ring (protection level) transitions, such as the fast system call instructions
- Instructions that explicitly affect the TLB or caches
- Software interrupt instructions
- I/O instructions
- Instructions within the virtualization extensions
- SMX instructions
- A range of privileged instructions such as HLT, MONITOR/MWAIT, RD/WRMSR, etc.

An important point is that our system does not require decoding and emulating general instructions, and to abort on one of these classes of instructions we need only decode an instruction sufficiently to identify its class. Any such decoding need happen only for the instructions within the transaction. Furthermore, many of these instructions are already detected in the VMM out of necessity (e.g., control and segment register updates, I/O instructions, virtualization instructions, most privileged instructions), and others can be readily intercepted without decoding (e.g. RFLAGS updates, debug registers, ring transitions, TLB instructions, software interrupts).

Exceptions: An exception on the core executing a transaction generally causes the transaction to abort, although the specification has variable clarity about which exception aborts are implementation-dependent and which are guaranteed. We assume that all exceptions that the Intel specification says "may" cause aborts "will" cause aborts. This behavior can easily be changed within our hardware model. The following exceptions cause aborts:

- All synchronous exceptions
- General interrupts
- NMI, SMI, IPI, PMI, and other special interrupts

As the VMM is already responsible for injection of general and special interrupts, it can easily detect aborts due to asynchronous exceptions. Detecting synchronous exceptions is slightly more challenging, as we discuss later.

Exception delivery within the context of a transaction abort has unusual, although sensible semantics. For synchronous exceptions, the abort causes the exception to be suppressed. For example, if a transaction causes a divide-by-zero exception, the hardware will abort the transaction, but eat the exception. For interrupts, the abort causes the interrupt to be held pending until the abort has been processed. For example, if a device interrupt happens during the execution of a transaction, the hardware will abort the transaction, and begin its fetch at `abort_label` before the interrupt vectors.

Resource limits: The specification indicates that transactions may only involve memory whose type is writeback-cacheable. Use of other memory types will cause an abort. Additionally, pages involved in the transaction may need to be accessed and dirty prior to the start of the transaction on some implementations. Finally, the specification warns against excessive transaction sizes and indicates that "the architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed."

Our interpretation of these parts of the specification is that a typical implementation is expected to be built on top of cache coherence logic. The implication is that transactions will behave differently on different hardware just to cache differences. The line size will likely define the conflict granularity for transactions. Two writes to the same cache line, but to different words, will likely conflict. Hence, the larger cache line, the more likely a transaction is to fall victim to an abort caused by a false conflict.

Our system allows the inclusion of a hardware model that can capture these effects, allowing the bullet-proofing of code that uses transactions, and the evaluation of the effects of different prospective hardware models on the code. Interestingly, because it is a software system, it creates the effect of hardware without resource limits.

A.1.2 Hardware lock elision (HLE)

RTM code is not backwards-compatible with older processors. Simultaneously, older code that uses locks needs to be rewritten in order to use RTM. Hardware lock elision is Intel's attempt to address both problems within the context of a transactional memory implementation. HLE and RTM are essentially different interfaces to a shared transactional memory implementation. With HLE, the hardware optimistically executes lock-based code using a transaction instead of a lock, automatically falling back on the original lock-based semantics if the transaction fails. The code itself is unaware of this operation.

HLE focuses on two existing x86 ISA features intended to facilitate the implementation of locks: the LOCK prefix and the XCHG instruction. In the following, we will focus on the XCHG instruction, which atomically exchanges the value stored at a memory location and the value stored in a register. LOCK-prefixed instructions operate similarly.[1]

[1]It is surprising that instructions such as XADD, which better support widely used ticket locks are not

Using the XCHG instruction, the following simplistic, yet illustrative lock/unlock primitives might be written. The + and - characters will be discussed later.

```
lock: // lock ptr in %rbx, will clobber %rax
  movq $1, %rax
spin:
  + xchg (%rbx), %rax
  cmpq %rax, $0
  jne spin
  ret
unlock: // lock ptr in rbx
  - movq $0, (%rbx)
  ret
```

HLE also introduces two additional instruction prefixes, XACQUIRE and XRELEASE, which are hints that the prefixed instruction is acquiring (releasing) a lock. To enable the above code to use HLE, we would replace the line marked as + with XACQUIRE (giving `xacquire xchg (%rbx), %eax`), and the line marked as - with XRELEASE (giving `xrelease movq $0, (%rbx)`).

These instruction prefixes overload the existing REPNE and REPE prefixes, which happen have no meaning for the particular instructions that HLE supports. On older processors without HLE, the presence of these prefixes on these particular instructions is *not* undefined, it is just ignored. That is, on older processors, the above code, behaves identically whether the XACQUIRE/XRELEASE prefixes are included.

currently supported by HLE, at least as far as the specification indicates.

On a processor with HLE, the XACQUIRE hint results in the core opening a transaction. The write to the lock essentially becomes an XBEGIN followed by a tagging of the lock's memory location in the cache. The value read by the XCHG is also stored. No write actually occurs. The XRELEASE hint then is effectively an XEND, with the added check that the value written back to the lock is the same as the one originally read during the XACQUIRE. In effect, the "lock" is now taken optimistically, and if this optimism is unwarranted at the XRELEASE, the transaction, from XACQUIRE to XRELEASE, is aborted. The abort is not software visible. Instead, the hardware reverts all the changes of the transaction, and then re-executes starting at the XACQUIRE, either trying the transaction again, or reverting to non-transactional processing (ignoring the XACQUIRE/XEND). In the end, it can always make progress if the original code was able to make progress.

A.2 Design and implementation

The implementation of our RTM emulation system is done in the context of our Palacios VMM [80], but its overall design could be used within other VMMs. We now describe our system, starting with the assumptions we make and the context of our implementation, followed by an explanation of the page-flipping approach the system is based on, and finally the architecture and operation of the system itself.

A.2.1 Assumptions

We assume that our system is implemented in the context of a VMM for x86/x64 that implements full system virtualization. Such VMMs can control privileged processor and machine state that is used when the guest OS is running, and can intercept guest manipulations of machine state. We assume the VMM infrastructure provides the following functionality for control and interception:

1. Shadow paging. We assume shadow paging in the VTLB model [68, Chapter 31] is available. It is not essential that the guest run with shadow paging at all times, merely that it is possible to switch to shadow paging during transaction execution.

2. Explicit VTLB invalidation. We assume that the VMM allows us to explicitly invalidate some or all entries in the shadow paging VTLB, independent of normal shadow page fault processing.

3. Shadow page fault hooking. We assume that the VMM allows us to participate in shadow page fault handling. More specifically, we assume it is possible for us to install a shadow page fault handler that is invoked after a shadow page fault has been determined to be valid with respect to guest state. Our handler can then choose whether to fix the relevant shadow page table entry itself, or can defer to the normal shadow page table fixup processing.

4. Undefined opcode exception interception. We assume the VMM allows us to intercept the x86/x64 undefined opcode exception when it occurs in the guest.

5. CPUID interception. We assume the VMM allows us to intercept the CPUID instruction and/or set particular components of the result of CPUID requests.

6. Exception interception. We assume the VMM allows us to selectively enable interception of exceptions and install exit handlers for them.

7. Exception/interrupt injection cognizance. We assume the VMM can tell us when a VM entry will involve the injection of exceptions or interrupts into the guest. If the

VMM uses guest-first interrupt delivery in which an interrupt can vector to guest code without VMM involvement, then it must be possible to disable this for the duration of the transaction so that the VMM can see all interrupt and exception injection activity.

The hardware virtualization extensions provided by Intel and AMD are sufficient for meeting the above assumptions. The same capabilities that the hardware provides could also be implemented in translating VMMs or paravirtualized VMMs. Common VMMs already meet 1, 2, 3, 5, and 7 as a matter of course. Items 4 (undefined opcode interception) and 6 (exception interception) are straightforward to implement. In AMD SVM, for example, there is simply a bit vector in the VMCB where one indicates which exceptions to intercept. On VM exit due to such an interception, the hardware provides the specific exception number.

Palacios already met most of the assumptions given in Section A.2.1. Shadow paging capabilities in Palacios reflect efforts to allow dynamic changes for adaptation. Palacios did not include support for assumptions 4 and 6 (exception interception). Perhaps ironically, our initial implementation of these two is for AMD SVM. However, Intel's VT also provides an exception bitmap to select which exceptions in the guest require a VM exit, so these changes could be readily made for VT. In Palacios, exception/interrupt injection cognizance (assumption 6) is implemented with a check immediately before VM entry, in SVM or VT-specific code. For the sake of initial implementation simplicity, we focused here again on the SVM version.

A.2.2 Palacios VMM

Our system is implemented in the context of our Palacios VMM. Palacios is an OS-independent, open source, BSD-licensed, publicly available embeddable VMM designed as part of the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University, the University of New Mexico, Sandia National Labs, and Oak Ridge National Lab. Detailed information about Palacios can be found elsewhere [79]. Palacios is capable of virtualizing large scale systems (4096+ nodes) with $< 5\%$ overheads [82]. Palacios's OS-agnostic design allows it to be embedded into a wide range of different OS architectures.

The Palacios implementation is built on the virtualization extensions deployed in current generation x86/x64 processors, specifically AMD's SVM and Intel's VT. Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that comprise the PC platform. Due to the ubiquity of the x86/x64 architecture Palacios is capable of operating across many classes of machines. Palacios has successfully virtualized commodity desktops and servers, high end Infiniband clusters, and Cray XT and XK supercomputers.

A.2.3 Architecture

Figure A.1 illustrates the architecture of our system. It shows a guest with two virtual cores, one executing within a transaction, the other not. The figure illustrates two core elements, the per-core MIME (Section A.2.4), which extracts fine-grain access information during execution, and the global RTME (Section A.2.5), which implements the Intel RTM model. The RTME configures the MIMEs to feed the memory reference information into

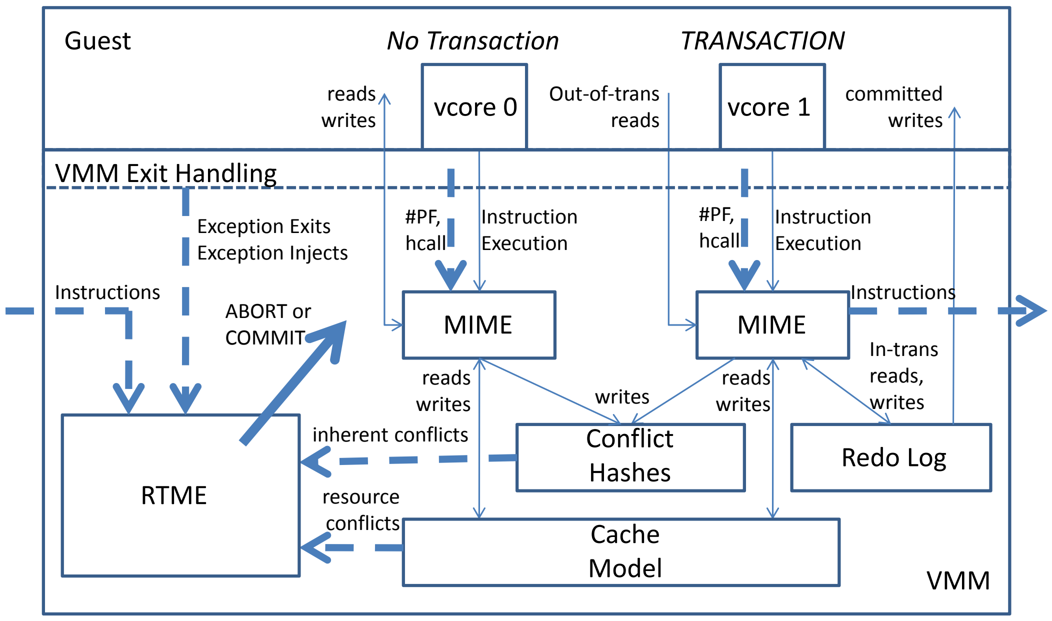


Figure A.1: Overall architecture of the system

the conflict hash data structures (for all cores), and the per-core redo log data structure (for each core executing in a transaction). The conflict hash data structures are used by the RTME to detect inherent memory access conflicts that should cause transaction aborts regardless of the hardware resource limitations. Additionally, the memory references feed a pluggable cache model, which detects hardware-limitation-specific conflicts that should cause transaction aborts. The RTME is also fed by the instruction sequences from cores operating in transactions, and by intercepted exceptions from the guest and injected exceptions or interrupts from the VMM, which also are needed to assess whether an abort should occur.

When no virtual core is executing in a transaction, we revert to normal execution of instructions by the hardware. The switch to the illustrated mode of operation occurs when an XBEGIN instruction is detected via an undefined opcode exception. Only this particular exception needs to be intercepted during normal (non-transactional) execution.

A.2.4 Memory and Instruction Meta-Engine

A core requirement of transactional memory emulation is being able to determine the memory addresses and data used by the reads and writes of individual instructions. When a transaction is active on any core, all cores must log their activities, producing tuples of the form `{vcore, sequencenum, rip, address, size, value, type}` where *sequencenum* orders the tuples of a given *vcore*, *rip* is the address of the instruction being executed, *address* is the address being read or written, *size* is the size of the read or write, *value* is the value read or written, and *type* indicates whether the reference is a read, write, or instruction fetch.

Our design accomplishes this fine-grain capture of the memory operations and data of instruction execution via the Memory and Instruction Meta-Engine, or the MIME. [2] One of the major contributions of this work is the novel page-flipping technique on which the MIME is based. This technique allows us to avoid instruction emulation and most aspects of instruction decoding. The MIME's page-flipping technique is based on the indirection and forced page faults made possible through shadow paging, and breakpoints to the VMM made possible through the hypercall mechanism.

Shadow paging and shadow page faults: It is necessary for the VMM to control the pages of physical memory that the guest has access to. Conceptually, with the VMM, there are two levels of indirection. Guest virtual addresses (GVAs) are mapped to guest physical addresses (GPAs) by the guest OS's page tables (the gPT), and GPAs are in turn mapped to host physical addresses (HPAs) by the VMM.

There are two main methods of supporting this mapping, nested paging and shadow

[2]The name is chosen for two reasons. First, the MIME mimics instruction execution. Second, it operates at a meta-level by manipulating the processor's instruction execution engine.

paging. In nested paging, the GPA$\rightarrow$HPA mapping is maintained in separate page tables constructed by the VMM (the nPT), and used by the hardware jointly with the guest page tables in the process of translating every memory reference in the guest. The VMM can safely ignore the gPT since the hardware is integrates the gPT and the nPT.

In shadow paging, the GVA$\rightarrow$GPA and GPA$\rightarrow$HPA mappings are integrated by the VMM into a single set of shadow page tables (the sPT) that express GVA$\rightarrow$HPA mappings that combine guest and VMM intent. The VMM makes the hardware use this integrated set of page tables when the guest is running. Unlike nested paging, the VMM cares about changes to the guest page tables and other paging state in shadow paging. Any architecturally visible change to guest paging state needs to invoke the VMM so that the VMM can adjust the integrated page tables to incorporate it. In order to do so, the VMM intercepts TLB-related instructions and control register reads and writes. Hence, any operation the guest performs to alert the hardware TLB of a change instead alerts the VMM of the change. The VMM's shadow paging implementation thus acts as a "virtual TLB" (VTLB) and the shadow page tables are the VTLB state which mirror what the guest "thinks" is happening in hardware, and map guest virtual addresses (GVA) to host physical addresses (HPA). Notably, this eliminates the need to convert from GVA to GPA. Since the guest writes to its guest page tables (gPT) and is unaware of the shadow page tables (sPT) additional synchronization must be performed, which we will illustrate with an example.

Suppose the guest creates a mapping (a page table entry) for the GVA `0xdeadb000`, which it does by writing this mapping to the gPT. The new mapping is not guaranteed to be architecturally visible until the TLB is informed. The guest does this by using an INVLPG instruction to flush any matching entry from the TLB. The VMM intercepts this instruction, where it informs the VMM that any entry that has the VTLB (the sPT) must be removed. When the guest later accesses some address on the newly mapped page, for example `0xdeadbeef`, the hardware walks the sPT, and on finding no entry, raises a page fault. The page fault is also intercepted by the VMM, which starts a walk of the gPT, looking for `0xdeadb000`. If no such entry existed in the gPT, the VMM would then inject a page fault into the guest. In this case, however, the gPT has a corresponding entry, and so the sPT is updated to include this entry, as well as a mapping to the appropriate HPA. Since the page fault occurred as a result of inconsistency between the sPT and gPT, it is referred to as a shadow page fault, and the guest OS is unaware that it ever happened. The next time the guest tries to access any address on the page `0xdeadb000` the sPT will have the correct mapping.

In the above example, a mapping was evicted from the VTLB (the sPT) due to the INVLPG instruction. It is also possible for VTLB eviction to be triggered for other reasons inside the VMM. In Palacios, there are internally usable functions for invalidating individual pages or all pages of an SPT. Thus, code in Palacios, such as the MIME, can force a shadow page fault to happen on the next access to a page.

Breakpoint hypercalls: In addition to forced shadow page faults, the MIME also relies on being able to introduce breakpoints that cause an exit back to the VMM, which we accomplish with a hypercall. Both AMD and Intel support special instructions, `vmmcall` in the case of AMD, that force an exit to the VMM. To set a breakpoint at a given instruction, we overwrite it with a `vmmcall`, after first copying out the original instruction To resume execution, we simply copy back in the original instruction content and set the instruction pointer to it.

Process: We now describe the MIME process for executing an instruction using the following example:

```
prev:  addq %rbx, %rax

cur:   INSTRUCTION

next:  movq %rdx, %rbx

...

target:

...
```

Here, `cur` is the address of the instruction we intend to execute, while `next` is the address of the subsequent instruction, and `target` is a branch target if the current instruction is a control-flow instruction.

Write-only data flow instruction: Let us make the current instruction more specific, for example, suppose it is

```
cur:   movq %rax, (%rcx)
```

This instruction writes the memory location in the register `%rcx` with the 8 byte quantity in the register `%rax`. MIME executes this instruction, and other instructions in the following way. We begin this process with the requirement that the sPT is completely empty.Note that the last step in the following reestablishes this for the next instruction.

1. We enter the guest with `%rip=cur`.

2. The instruction fetch causes a shadow page fault, which exits back to the VMM, which hands it to the MIME.

3. The MIME discovers this is an instruction fetch by comparing the faulting address and the current `%rip` and noting the fault error code is a read failure. In response,

it creates an sPT entry for the page the instruction is on. While the page is fully readable and writable by the MIME, the sPT entry allows the guest only to read it. The MIME then overwrites `next` with a hypercall, saving the previous content.

4. We enter the guest with `%rip=cur`.

5. The instruction fetch now succeeds. Instruction execution now succeeds as well, up to the data write. The data write produces a shadow page fault, which exits back to the VMM, which hands it the MIME.

6. The MIME discovers this is a data write by noting that the fault code is a write failure. It can optionally compare the fault address with the instruction pointer to determine whether this is an attempt to modify the currently executing instruction. This can serve as a trigger for transaction abort when the MIME is used in the TM system.

7. In response to the data write, the MIME maps a temporary staging page in the sPT for the faulting address, and it stores the address of the write.

8. We enter the guest with `%rip=cur`.

9. The instruction fetch and the data write now succeed and the instruction finishes, writing its result in the temporary staging page.

10. `%rip` advances to `next`, resulting in the fetch and execution of the hypercall (note that the code page is now mapped in), which exits back to the VMM, which hands it to the MIME.

11. The MIME now reads the value that was written by the instruction on the temporary staging page. It can now make this write available for use by other systems. For

example, the RTM system will place it into its own data structures if a transaction is occurring. If no other system is interested, it copies the write back to the actual data page.

12. At this point the MIME has generated two tuples for the record: the instruction fetch and the data write.

13. The MIME now restores the instruction at `next`

14. The MIME invalidates all pages in the VTLB. Strictly speaking, only two pages are unmapped from the sPT, the code page and the temporary staging page.

15. If MIME-based execution is to continue with `next`, goto 1, otherwise we are done.

Read-only data flow instruction: For an instruction like

```
cur:   movq (%rcx), %rax.
```

which reads 8 bytes from the memory location given in `%rcx` and writes that result into the register `%rax`, execution is quite similar. At stage 5, a shadow page fault due to the data read will occur. In stage 6, the MIME will detect it is a data read and sanity check it if needed. In stage 7, the MIME will map the staging page read-only, and copy the data to be read to it. This data can come from a different system. For example, the RTM system might supply the data if the read is for a memory location that was previously written during the current transaction. After the instruction finishes, it will then provide two tuples for the record: the instruction fetch and the data read.

Read/write data flow instruction: It is straightforward to execute an instruction such as

```
cur:   addq %rax, (%rcx)
```

which reads the 8 bytes from memory at `%rcx` adds them to the contents of `%rax` and then writes the 8 byte sum back to the memory at the address in `%rcx`. For all but the final write, the execution is identical to that of the read-only instruction given above. After completing stage 7, the staging page will be mapped read-only, and thus there will be an additional shadow page fault corresponding to the write. This fault will be handled in the same manner as with the write-only instruction. After the instruction finishes execution, it will then provide three tuples for the record: the instruction fetch, the data read, and the data write.

Control flow instruction: If a control flow instruction reads data (e.g., an indirect jump) or writes data (e.g., a stack write on a call), these reads and writes are handled in the same manner as the preceding data flow instructions. Since all the conditions to be checked (e.g., flags) are known at this point, we can "emulate" the instruction, placing the hypercall at the jump target.

The key difference in the processing of control flow instructions is that there are up to two breakpoints that need to be introduced. For unconditional control flow, a single breakpoint needs to be introduced at the target address instead of at the next instruction. For conditional control flow, two breakpoints need to be introduced, one at the target and one at the next instruction.

Generalization: Although the above description uses simple two operand instructions and the simplest memory addressing mode as examples, it's important to note that the technique works identically for different numbers of operands and for arbitrary addressing modes. Indeed, even for implicit memory operands, the hardware will produce shadow page faults alerting us to their presence. The primary limitation is that an instruction with multiple reads and/or multiple writes to the same page may not have all of its reads and writes captured. We describe this in detail later. All addressing mode computations, as well as segmentation, are done well before a page fault on instruction or data references can result. The hardware does this heavy-lifting.

Instruction decoding and emulation: Step 3 of the processing described above requires basic instruction decoding. The issue is that x86/x64 instructions are of variable length (from 1 to 15 bytes). Hence, in order to determine what `next` is, we need to be able to determine the size of the current instruction. If the MIME does not need to trace control-flow instructions, this is the only requirement. If control-flow instructions are to be handled by the MIME, then we must further decode control-flow instructions to the point where we can also determine their target address. Our implementation uses the open source Quix86 decoder [76] to do this decoding for us. No emulation is done at all—we rely on the hardware to do instruction execution for us instead.

Page-flipping in Palacios: Since transactional memory keeps track of memory at much finer granularity than locks, we needed to be able to record all reads and writes happening in the guest. We leverage shadow paging and control over the sPT in order to accomplish this.

At the beginning of a transactional block, marked by an XBEGIN, we set up the necessary internal state, and flush the sPT. This means that each memory access attempt will result in a page fault, which we can catch in the VMM. The page-flipping technique clears the sPT between instructions, which allows us to use shadow page faults to track all of the reads and writes that happen in the guest, the process is illustrated in Figure A.2. Once the sPT has been flushed, the first page fault that should occur in the guest is a result of the IFETCH of the instruction immediately following the XBEGIN, which we verify by checking that the faulting address is the same as the address pointed to by the guest's

RIP. We allow the VMM to map this memory location in, and await the memory accesses caused by the instruction. Since x86 instructions can read from memory, write to memory, do both, or neither, we must be prepared to catch any of these events. If we see a read from memory, we map in the corresponding page, map it as read only, and allow the instruction to restart. If we see a write to memory, we map the page in, mark it with the same writable status as in the gPT, and allow the instruction to restart. We do not automatically mark the page as writable to keep from writing to pages the guest had marked as read-only. If no reads or writes occur, the sPT is once again flushed in order to catch the IFETCH of the next instruction.

Read optimization in RTM: In our earlier description of handling read-only instructions and read-write instructions, we describe the use of a staging page during the read—when a data read is detected, we copy the value to be read to a staging page and present this page to the guest. In RTM, this is required when a core executing a transaction reads a value it has previously written during the transaction. At this point, in order to maintain isolation, the written value exists only in a redo log and must be copied from it. For a core that is not executing in a transaction, or for an in-transaction read of a value that was not previously written in the transaction, the staging page can be avoided and the read allowed to use the actual data page. This optimization is included in our RTM system.

Implementation limitations: There are two limitations of our implementation of the MIME that we are aware of. The first is that in the context of a single instruction, we can detect only the first read and the first write to an individual page. The reason for this limitation is that in order to make progress in instruction execution, we must resolve page faults. On a shadow page fault on a read of page, we enable read access to the page, and similarly, a write enables write access to the page. Because the enabling access is done at the page granularity, subsequent references to the page do not result in further shadow page faults. The primary class of instructions where this may be an issue is string instructions with a REP prefix. The MIME can detect this prefix and raise an error to the user—for RTM, the transaction is aborted. Many VMMs, including Palacios, already have instruction emulation for this class of instructions, as it is often needed for I/O and memory hooking. In principle, MIME could fall back on emulation for this specific case, although it does not do so currently.

The second implementation limitation is that instructions or data that span two pages are not supported. On the x86/64 the only alignment requirement imposed by the hardware is byte-alignment, so such cases are legal. However, compilers try very hard to avoid producing such cases, as unaligned references may be more expensive. Currently MIME detects this situation and simply raises an error to the caller. The RTM implementation turns it into a transaction abort.

A.2.5 Restricted Transactional Memory Engine

One of the benefits of our page-flipping technique is that we are able to accomplish single-stepping of the guest OS without having to emulate any instructions except for XBEGIN, XEND, and XABORT. When we begin a transaction, we begin page-flipping, but need to be able to return to the VMM once an instruction has finished running. To do this, we decode the instruction we are about to run to find its length, and replace the following instruction with a VMMCALL with a value specific to transactional memory. The entire process is captured as a finite state machine, illustrated in Figure A.3.

As shown in Figure A.1, the RTME uses per-virtual core MIMEs to capture instructions and their memory accesses in a step-by-step manner during execution. The only in-

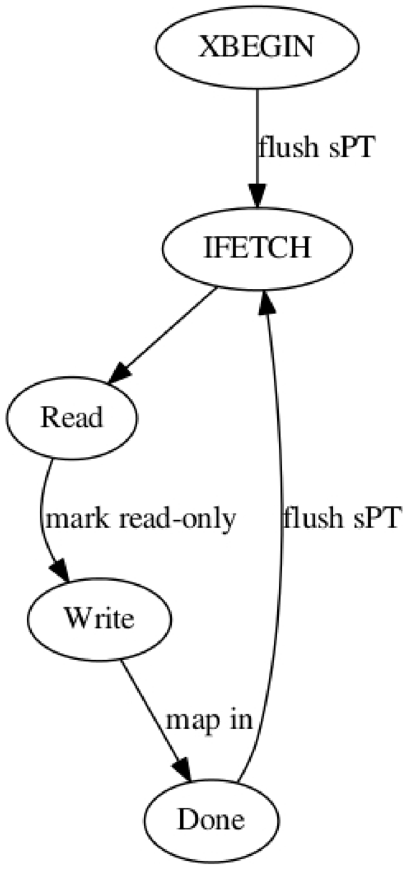


Figure A.2: The page-flipping technique

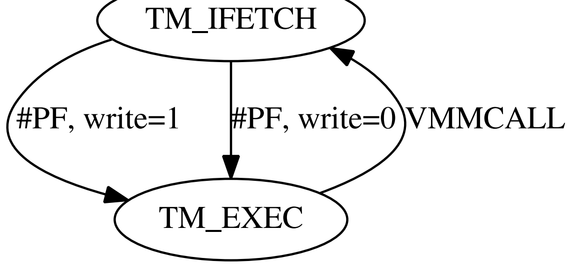


Figure A.3: TM state machine

structions it needs to emulate are the XBEGIN, XEND, XABORT, and XTEST instructions. Because these instructions are not available in the hardware, they cause an undefined opcode exception which is caught by the VMM and delivered to the RTME.

The initial XBEGIN is emulated by capturing its abort target, advancing RIP to the next instruction, and switching all virtual cores to MIME-based execution. Additionally, the RTME has the VMM enable exiting on all exceptions with callbacks to the RTME. A special check is enabled in the interrupt/exception injection code which is run before any VM entry. This check tests if an injection will occur on entry, and if so it invokes a callback to the RTME before the entry. Either callback is interpreted by the RTME as requiring a transaction abort for that virtual core.

From the next entry on, MIME-based execution occurs on all virtual cores. On all virtual cores, the writes seen by the MIME are written to the conflict hash data structures. For a virtual core that is not executing in a transaction, the writes are also reflected to guest memory, and all reads are serviced from guest memory. For a virtual core that is executing in a transaction, writes are sent to the redo log instead of to guest memory. Reads are serviced from guest memory, except if they refer to a previous write of the transaction, in which case they are serviced from the redo log. For all cores, reads and writes are also forwarded to the cache model by the RTME.

In addition to the callbacks described earlier, the RTME is also called back by the MIME as it executes its state machine. This allows the RTME to examine each instruction and its memory operations to see if an abort is merited. Instructions are checked against the list given earlier. For all memory operations, the RTME checks the conflict hash data structures and the cache model. The former indicates whether a conflict would have occurred assuming an ideal, infinite cache. For example, if this core is not in a transaction, and has just written a memory location that some other core that is in a transaction previously wrote, a conflict is detected and the other core needs to abort its transaction. The cache model determines if a conflict due to hardware limitations has occurred. For example, if the current write is coming from a core that is executing in a transaction, and that write would cause a cache eviction of a previous write from the transaction, the cache model would detect this conflict and indicate that the current core needs to abort its transaction. A final source of an abort is when the RTME detects the XABORT instruction during the MIME instruction fetch.

Handling a transaction abort is straightforward: the writes in the redo log are discarded, the relevant error code is recorded in a guest register, the guest RIP is set to the abort address, and the guest is re-entered. Transaction commits occur when the XEND instruction is detected, and are also straightforward: the RTME plays the redo log contents into the guest memory, advances the RIP, and re-enters the guest. For either an abort or a commit, we also check if it is the last active transaction. If so, we switch all cores back to regular execution (turning off MIME, callbacks, and exception interception, except for the illegal opcode exception, which is needed to detect the next XBEGIN).

XTEST instructions are identified by the RTME through a UD exception, if the instruction is run during an active RTM section, then the ZF flag is set, otherwise it is cleared.

Redo log considerations: Our redo log structure is not, strictly speaking, a log. Rather, it stores only the last write and read to any given location. However, during MIME execution, there exist short periods where the most recent write or read is actually stored on the MIME staging page. A versioning bit is used so that when the MIME-based execution of an instruction completes, it is possible to update the redo log with newer entries on the staging page. These aspects of the design allow us to compactly record all writes and internal reads of a transaction.

Conflict detection: In addition to the conflict hashes, conflict detection in the RTME uses a global transactional memory (TM) state, a global transaction context, a per-core TM state, and a per-core transaction number. The global TM state indicates whether any core is running a transaction, while the per-core TM state indicates whether the specific core is executing a transaction. Each core assigns sequence numbers to its transactions in strictly ascending order, and the per-core transaction number is the number of the current, or most recent transaction on that core. The global transaction context gives the number of the currently active transaction, or most recently completed (aborted or committed) transaction on each core.

When any core is running a transaction, all cores must record the memory accesses they make, we accomplish this through the use of two hash tables. The first, called the address context hash, is a chained hash mapping memory locations to timestamped accesses. Each entry in the hashed bucket represents the global transaction context at the time of a memory operation, which acts as a ordering, or timestamp. In this way we are able to both record all memory accesses done by a core, as well as keep track of when they occurred. Since all memory accesses are tagged with the global context, when a core is checking for conflicts it can simply look at accesses made with the same context as its current transaction number. Entries in the hash have the form `{addr : (global_ctxt)→(global_ctxt)→...}`

The second hash table, called the access type hash, keeps track of the type of memory operation that was run on an address in a given context (read, write, or both). When a memory operation is run by a core, it creates one entry for each core in its hash. Data is duplicated in this manner to facilitate quick lookup on conflict checking as well as garbage collection. Entries in this hash have the form `{addr|core_num|t_num :  access_type}`

Suppose we are running on a guest with two virtual cores, and core 0 begins a transaction. Each core will begin running its MIME and recording its memory accesses. Now suppose core 1 runs an instruction which writes to memory address `0x53`. It will first note the global transactional context, and add a node to the bucket for address `0x53` in the address context hash with this context. It will then make two new entries for the access type hash, one for each core in the system. Each entry will map address `0x53`, a core number, and that core's transaction number to a structure indicating the access was a write.

Suppose we are running a guest with two virtual cores: 0 and 1. Core 0 begins a transaction, and the global context indicates that core 0 is on transaction number 5, core 0 on number 3. Core 1 runs an instruction that writes to memory location `0xdeadbeef`, it records this fact by first inserting into the address context hash a mapping from the location `0xdeadbeef` to the value `5:3`, indicating the context during which the write happened. It then adds mappings to the access hash for keys `0xdeadbeef:core 0:5` and `0xdeadbeef:core 1:3`, with values indicating those accesses were writes.

The conflict hashes data structure consists of two hash tables, for address context and access type. The address context hash maps from a memory address to a list of transaction contexts in which it has been seen. That is, when an access to address N occurs, we hash N to its list, and append a copy of the current global transaction context to the xlist. The transaction contexts effectively are timestamps of the references. The access type hash maps from a concatenation of the memory address, the core, and that core's transaction number to the type of access that was seen (read, write, both). Note that this is not a list, but rather simply records whether the address was read, written, or both. We illustrate how recording happens with an example.

	External Read	External Write
Read	Continue	Abort
Write	Abort	Abort

Figure A.4: The results of a transaction given the combination of a transaction operation and an external operation on the same memory address.

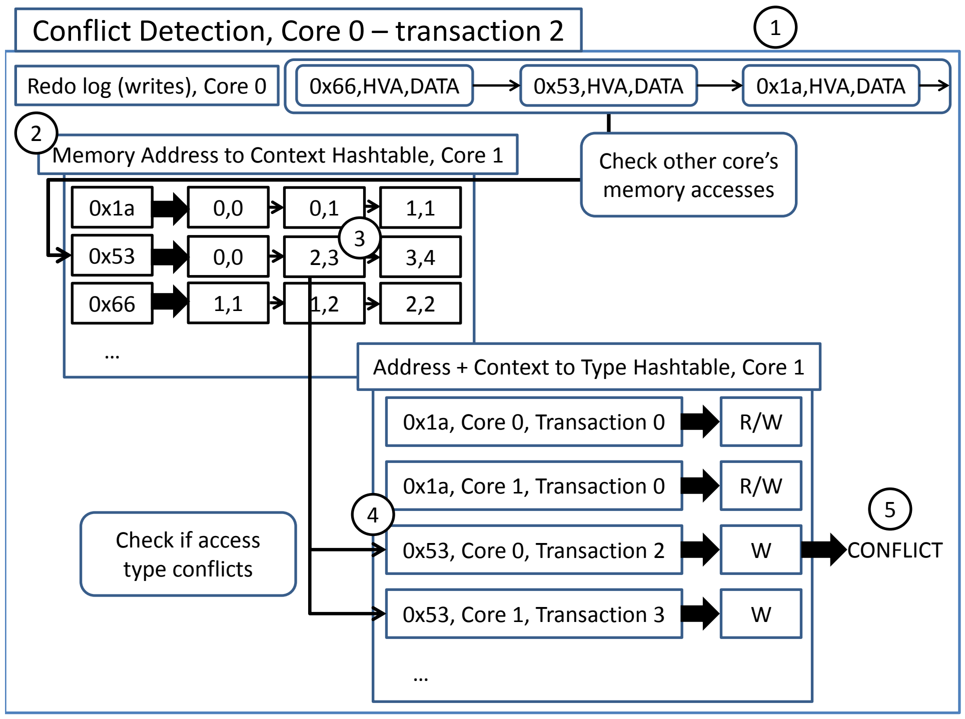

Conflict Detection, Core 0 – transaction 2
1
Redo log (writes), Core 0
0x66,HVA,DATA
0x53,HVA,DATA
0x1a,HVA,DATA
2
Memory Address to Context Hashtable, Core 1
Check other core's memory accesses
0x1a
0,0
0,1
1,1
0x53
0,0
2,3
3
3,4
0x66
1,1
1,2
2,2
...
Address + Context to Type Hashtable, Core 1
0x1a, Core 0, Transaction 0
R/W
0x1a, Core 1, Transaction 0
R/W
Check if access type conflicts
4
0x53, Core 0, Transaction 2
W
5
CONFLICT
0x53, Core 1, Transaction 3
W
...

Figure A.5: An example of a write/write conflict detected by core 0 on a write from core 1

If the memory accesses of a core executing a transaction conflict with those of any other core, the transaction must be aborted. To check for conflicts, we use the context hashes. Conflict checking could be done after each instruction, or when attempting to commit a transaction. In our implementation, conflict checking is cheap relative to instruction run time and so we generally do it after each instruction.

In Figure A.5 we illustrate the process of conflict detection. At the end of an instruction in a transactional block, core 0 walks its redo log of writes, shown as step 1. In step 2, core 0 checks if any conflicting memory accesses have been made by looking at every other core's address context hash. In the figure, core 0 checks for conflicting accesses to memory address `0x53`. In step 3, core 0 finds an entry for `0x53` in core 1's address context hash, and walks the list of contexts during which core 1 accessed `0x53`. Core 0's current transaction number is 2, so the entry made during context `{2,3}` is a potential conflict. In step 4 core 0 checks the entry for address `0x53` in core 1's access type hash to identify the kind of access made. Having found that core 1 wrote to address `0x53` when core 0 was in transaction 2, a conflict is detected, and core 0 must abort its transaction, shown as step 5.

When the MIME indicates that an instruction executing in a transaction is finished, the RTME, executing on that core will scan its redo log, checking each read and write for conflicts with each of the other cores. Suppose now that core 0 is scanning its redo log, and it contains a write to address `0x53`. Core 0 will check the entry for address `0x53` in every other core's address context hash. If it finds a node with the same context as its own current transaction number, that implies a potential conflict, and core 0 will then check the entry in that remote core's access type hash for the key with the address `0x53` and core 0's transaction number under which the access was made. This tells it what kind of access the other core performed, and thus we can discard a read/read pair as non-conflict.

We now see the beauty of the access type hash table - a core knows which core number it is, and it knows its current transaction number, so for each entry in its redo log, it only has to check one hash table entry in each other core. This gives conflict checking a runtime of

$$Runtime = O((num\_cores \times hash\_lookup) \times redo\_log\_entries)$$

and since hash lookup is constant time, and the number of cores is constant, the runtime is proportional to the size of the redo log.

Garbage collection: Memory use expands during execution as the redo logs and conflict hashes grow in size. Redo logs are garbage collected at the end of each transaction. Since the conflict hashes contain information from multiple generations of transactions, we must answer the questions of how to determine which entries are garbage, and when to perform this garbage collection.

Garbage collection leverages the access type hash. We start by noting the current global transaction context, then we iterate over all the keys of the access type hash, and for each key (an address), we walk over its corresponding list of contexts. If we find a context that is strictly less than the global context, this means that there is no core left that may need to check that memory operation, as it happened during transactions that are no longer active. We can generate from this stale context the corresponding keys for the access type hash, and delete those keys from it. Finally, we delete the stale context from the list, and delete the key from the address context hash if the list is now of zero length. A locking strategy is employed to assure that a garbage collection and MIME accesses are mutually exclusive.

When to garbage collect is a more difficult question, as when we have an opportunity to do so, we cannot be certain about the state of other cores, or when the next opportunity to collect may occur. Currently in our implementation each core will garbage collect on every transaction completion.

Core separation: A major aspect of our implementation is to allow each virtual core to run independently of all other virtual cores to the greatest extent possible, as in a real system. When a transaction is active, each core is independently being executed by a MIME, and does not know, or care, what state any of the other cores is in. Whether other cores are recording, or running their own transactions, a core will operate in the same way. Garbage collection is also handled independently of any other cores behavior. The only time cores interact with one other is when setting the system in TM mode, or checking for conflicts. Ironically enough, during conflict checking and garbage collection, cores must acquire locks on the data s structures they wish to utilize.

Interrupt injection: MIME-based execution of guest instructions during a transaction operates considerably slower than direct hardware execution. As a consequence, in our system, a transaction is more likely to experience an external interrupt and thus a transaction abort. Of particular note are timer interrupts, which are ultimately derived from host time. It is important to note that all external interrupts that the guest sees are injected by the VMM. Hence, it is always possible to delay interrupt injection until after a transaction completes. Furthermore, Palacios has a time-dilation feature [16], modeled on DieCast [55], that was originally designed for interfacing with external simulators. Time-dilation can also be used to slow the apparent passage of time in the guest by manipulating guest time sources, including the rate or period of timer devices (such as the APIC timer) that produce interrupts. Using interrupt injection delay and time dilation, it is possible to execute the transaction without an apparent interrupt, or with an interrupt probability similar to what would have been experienced if MIME-based execution were as fast as hardware execution.

If the transaction is aborted for any reason, the redo log gets cleared, and execution continues at the `fail_call` with no changes to memory having been made. If the transaction reaches an XEND, the transaction has finished successfully, and the VMM will walk over the redo log, and “commit” the values of writes that had occurred during the transaction to memory.

A.3 Evaluation

We considered three factors when evaluating our RTM implementation: its size, how it runs code with transactions, and the performance of the implementation relative to the native execution rate of the hardware and compared to a different emulator.

Test environment: All testing was done on a Dell PowerEdge R415. This is a dual socket machine with each socket having a quadcore AMD Opteron 4122 installed, giving a total of 8 physical cores. The machine has 16 GB of memory. It ran Fedora 15 with a 2.6.38 kernel. Our guest environment uses two virtual cores that run a BusyBox environment based on Linux kernel 2.6.38. The virtual cores are mapped one-to-one with unoccupied physical cores. This machine does not have an HTM implementation.

Implementation size: Our implementation of RTM emulation is an optional, compile-time selectable extension to Palacios, and we made an effort to limit changes to the core of Palacios itself. There were two major areas where we had to modify the Palacios core, namely (1) handling of exceptions and interrupts, some of which are needed to drive the RTME, and (2) page fault handling, allowing some page faults to drive the MIME. These changes and the entirety of the extension code comprise 1300 lines of C. Given the size and very clear changes to the Palacios codebase, it should be possible to port our implementation to other VMMs.

Test cases: To test the correctness of our implementation, we needed a test suite which would present the implementation with various behaviors, and an ability to test the outcome. GCC 4.7 includes support for compiling the Haswell transactional instructions, but the test cases shipped with it only evaluate the behavior of software transactional memory. We found we had to write our own test cases, which test the following scenarios: (1) transaction calls XABORT after making no changes to memory, (2) transaction calls XABORT after having “written” to memory, (3) transaction writes memory with an immediate value, (4) transaction reads memory into a register, (5) transaction writes a register to memory, (6) transaction reads and writes the same memory location, (7) transaction thread writes to distinct, addresses, and (8) transaction and non-transactional thread write to overlapping addresses. The test cases are written using pthreads. After the threads set their affinity for distinct virtual cores, and synchronize, they then repeat their activity. Hence, over time, various possible orderings of execution are seen, as are aborts due to external causes (e.g., interrupts). These test cases form a “correctness test” of our implementation, which it passes.

Performance: The MIME-based execution model must obviously be slower than normal execution under the VMM or at native speeds. To quantify this slowdown over native, we ran an additional microbenchmark on our system and on a new, first-generation Haswell machine, an HP Proliant DL320e with a single-socket, quad-core Intel Xeon E3-1720v3 and 8GB RAM.

This benchmark consists of one thread pinned to a single core that enters a transaction, writes to a memory location, and then exits the transaction. The benchmark measures the time spent running 10 such transactions, and is intended to typify a common transactional code path. We averaged this runtime over 100 runs. To ensure accurate timing for RTM emulation, we used the TSC in passthrough mode to measure elapsed time. We found that on native, the average time spent running the 10 transactions was 2.57usec, while under MIME, the average time was 853.88usec.

Future work on transactional memory emulation will include comparing performance of multi-threaded applications, more complicated transational semantics including transaction restarts, and testing transactions that the hardware would not be able to support, such as those that exceed architectural limits.

We also ran our test cases on Intel's Software Development Emulator (SDE), where we found a slowdown on the order of 90,000×—our RTME runs approximately 60 times faster during a transaction. Moreover, the SDE's overhead occurs all the time, as one might expect for full emulation. There is a caveat in these numbers, however. The cache model we are using in our RTME is the null model (no aborts due to hardware resource limits), while Intel's is not. That said, we found that the average MIME "step" – the average time to process a read or write – took on the order of 7,000 cycles. This means that we would need to use a cache emulator that took >400,000 cycles per read or write in order for our system to slow down to speeds of emulation.

A.4 Conclusions

We developed an implementation of Intel's HTM extensions in the context of a VMM using MIME, a novel page-flipping technique. Our implementation allows the programmer to write code with TSX instructions, allows for bullet-proofing of code for various hardware architectures, as well as allowing tight control of the environment under which a transaction is occurring. We are able to achieve this with limited instruction decoding, and at speeds approximately 60 times faster than under emulation.

Appendix B

VMM/Architecture simulator integration

In this appendix I describe work that I did in creating a service for automated translation between checkpoints of GEM5 and Palacios virtual machines. This work was done in the Summer of 2014. GEM5 [13] is an architecture simulator, capable of emulating a processor down to its microarchitecture, which is very useful for architecture research. However, due to its detail and complexity, it is also extremely slow, in some cases 3 orders of magnitude slower than native execution. By providing a mechanism for translation of checkpoints, researchers would have the opportunity to transition to detailed GEM5 state only when necessary, to observe detailed architectural behavior, and remain in the much faster virtualized state for the rest of execution. Similar approaches have been recently researched and considered directly inside the GEM5 simulator [107]. The following is the contents of the README we have provided as part of the snapshot of our tool, GEM5 code state, and Palacios code state.

Previous work for this project was done as a part of Northwestern University class EECS 441: Resource Virtualization taught by Prof Peter A. Dinda, and completed by Madhav Suresh, John Rula, and George Tziantzioulis. I am thankful to them for their initial efforts, as well as guidance that was provided as I continued work on this idea.

What follows is the contents of the README file that is included with a snapshot of the code necessary for checkpoint translation, as well as snapshots of both Palacios and Gem5 codebases in the states that the translation counts on. The translation depends on the behavior to be such as it is in these snapshots, which includes accounting for any bugs that may exist in either or both codebases.

README This directory contains a snapshot of code for doing migration of virtual machines between Palacios and GEM5. This is an involved process and our tools are a proof of concept, hence the release of this as a snapshot. Minor changes to the Palacios codebase have also been commit and pushed to the devel branch. No changes to GEM5 are needed, but we include a copy of the GEM5 tree we use for testing just in case. The migration tools rely on the relevant bugs/features in this version of GEM5 and may not work with any other.

By "migration of virtual machines between Palacios and GEM5", we mean the following:

1. Creation of a pair of VMs based on identical images, one for GEM5, one for Palacios.
2. Transformation of a Palacios checkpoint into a GEM5 checkpoint. This allows you to checkpoint a VM in Palacios and resume it in GEM5.
3. Transformation of a GEM5 checkpoint into a Palacios checkpoint. This allows you to checkpoint a VM in GEM5 and resume it in Palacios.

The set of devices that can be transformed is limited (generally to a subset of devices used in GEM5 (PIC, PIT, APIC, IOAPIC, SERIAL, ...).

B.0.1 Files

The files included as part of the tool snapshot are:

- arch-snapshot.tgz
  - This contains the migration tools and numerous working examples ("pairs" of Palacios VMs and GEM5 VMs)
  - Migration tools are compact, the working examples are not
  - This also includes a snapshot of Palacios configured and built as appropriate
- gem5-snapshot.tgz
  - Snapshot of the version of GEM5 and its build that we are using

B.0.2 Documentation for checkpointing translation process

Requirements:

- Palacios devel commit `76fea3b8a640b9b1a509b6ad20a2868ced5e5548`
- Gem5 changeset `8592:30a97c4198df`
- Gem5/Palacios kernel pair (must have a serial stream console)

Checkpointing and restoring virtual machines is accomplished via the following steps:

1. Make Gem5 skeleton (create m5.p pickle file)

```
cd /path/to/gem5-base

source ENV

cd gem5
```

```
./build/X86_FS/m5.opt -d ./m5out/ configs/example/fs.py --kernel="pair1"

nc localhost 3456

    m5 checkpoint

cp m5out/cpt.CURRENT/m5.cpt /path/to/xlation/dir

mv m5out/cpt.CURRENT m5out/cpt.1

cd /path/to/xlation/dir

./m5parse.py m5.cpt m5.p
```

2. Palacios Boot (create binary memory image and text checkpoint)

```
v3_init

export PATH=/path/to/palacios/linux_usr:$PATH

v3_create -b /path/to/your-kernel-pair-config.xml go

v3_launch /dev/v3-vmX

v3_stream /dev/v3-vmX streamY

(optional) make some fs changes

v3_pause /dev/v3-vmX

v3_guest_mem_access /dev/v3-vmX read 0 MEM_SIZE_BYTES > /path/to/mem_save

v3_save /dev/v3-vmX KEYED_STREAM textfile:/path/to/cpt 1
```

3. Pal $\rightarrow$ Gem Translation (create p25out/m5.cpt file)

```
cp -r /path/to/cpt /path/to/xlation/tpair

./p2m.py -v > /path/to/p2m.output
```

4. Restore in Gem5 (create m5.cpt checkpoint and gemmem memory image)

```
cp p25out/m5.cpt /path/to/gem5/m5out/cpt.1/
cp /path/to/mem_save /path/to/gem5/m5out/cpt.1/system.physmem.physmem
cd /path/to/m5-base
source ENV
cd gem5
./build/X86_FS/m5.opt -d ./m5out/ configs/example/fs.py --kernel="pair1"
    -r 1
nc localhost 3456
    m5 checkpoint
mv m5out/cpt.NEW /path/to/gem5_save
zcat /path/to/gem5_save/system.physmem.physmem >
    /path/to/xlation/dir/gemmem
```

5. Gem → Pal Translation (make tmp/* palacios checkpoint files)

```
cp /path/to/gem5_save/m5.cpt /path/to/xlation/dir
mkdir tmp
./m2p.py -v > /path/to/m2p.output
```

6. Restore in Palacios

```
v3_create -b /path/to/your-kernel-pair-config.xml go
v3_guest_mem_access /dev/v3-vmZ write 0 MEM_SIZE_BYTES <
    /path/to/xlation/dir/gemmem
cp /path/to/xlation/dir/tmp/* /path/to/xlation/dir/tpair
v3_load /dev/v3-vmZ KEYED_STREAM textfile:/path/to/xlation/dir/tpair 1
v3_launch /dev/v3-vmZ
```

```
   v3_stream /dev/v3-vmZ streamY
```

All files are available via `http://v3vee.org/palacios/`