NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science
Department

Technical Report
Number: NU-EECS-15-02

May 11, 2015

Prospects for Shaping User-centric Mobile Application Workloads to Benefit the Cloud

Maciej Swiech, Huaqian Cai, Peter Dinda, Gang Huang

Abstract

Approaches to making cloud operation more efficient, for example through scheduling and power management, largely assume that the workload offered from mobile, user-facing applications is a given and that the cloud must simply adapt to it. We flip this assumption 180 degrees and ask to what extent can we instead shape the user-centric workload into a form that would benefit such approaches. Using a toolchain that allows us to interpose on frontend/backend interactions in popular Android applications, we add the ability to introduce delays and collect information about user satisfaction. We conduct an “in the wild” user study using this capability, and report on its results. Delays of up to 750 ms can be introduced with little effect on most users, although this is very much user and application dependent. Finally, given our study results, we consider reshaping the application requests by selective delays to have exponential interarrival times (Poisson arrivals), and find that we are often able to do so without exceeding the user's delay tolerance.

This project is made possible by support from the United States National Science Foundatation (NSF) via grant CNS-1265347, and by a gift from Intel Corporation.

Keywords mobile environments, user studies, application studies, power management, empatic systems

Prospects for Shaping User-centric Mobile Application Workloads to Benefit the Cloud

Maciej Swiech
Northwestern University

Huaqian Cai
Peking University

Peter Dinda
Northwestern University

Gang Huang
Peking University

Abstract

Approaches to making cloud operation more efficient, for example through scheduling and power management, largely assume that the workload offered from mobile, user-facing applications is a given and that the cloud must simply adapt to it. We flip this assumption 180 degrees and ask to what extent can we instead shape the user-centric workload into a form that would benefit such approaches. Using a toolchain that allows us to interpose on frontend/backend interactions in popular Android applications, we add the ability to introduce delays and collect information about user satisfaction. We conduct an "in the wild" user study using this capability, and report on its results. Delays of up to 750 ms can be introduced with little effect on most users, although this is very much user and application dependent. Finally, given our study results, we consider reshaping the application requests by selective delays to have exponential interarrival times (Poisson arrivals), and find that we are often able to do so without exceeding the user's delay tolerance.

1 Introduction

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 paper, 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.

As an example, consider making a modern user-facing cloud application such as Pinterest, Pandora, or Google Translate 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 data centers and the network that are the physical underpinnings of the cloud. User interactions with the frontend result in requests to the backend. The data center brings up or shuts down hardware to accommodate the offered load of the requests from the application's users. The assumption given above 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 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.

In this paper, we consider the prospects for relaxing the assumption. 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. In effect, we consider shaping the user-driven workload analogously to how a packet stream might be shaped at entry to a network. 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 [13, 23, 32, 17, 24], and that its is possible to take advantage of it in those contexts [18, 19, 20, 21, 22, 29, 36, 16, 17, 31, 30].

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 to the user experience and collected both the user feedback and the individual requests going from the frontend to the backend. 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.

We then consider a system which selectively delays requests going from frontends to the backend so as to shape the arrival process at the backend 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. The system we consider also can limit the overall arrival rate (throttling). We simulate this system 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, rate limits) 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 while not introducing delay that leads to dissatisfaction.

The contributions of our work are:

- We introduce the concept of shaping user-driven requests originating from the frontend of a mobile application and going to the cloud backend to meet goals established by the backend. We refer to this concept as *user traffic shaping*.

- 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 room to do user shaping by introducing delays in the request stream. The tolerance for introduced delay exists across the whole subject group, and it varies across individuals.

- We describe a potential system that uses this room and varying tolerance to introduce delays that shape the user request stream so that it is closer to a Poisson process, and can be rate limited.

- We evaluate this system, 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.

2 Related Work

Achieving an effective and responsive user experience is critical to the success of cloud applications, but at the same time there is a growing interest in making the data centers that their backends run on more efficient and sustainable. By late 2011, Google's data center operation involved 260 megawatts of continuous power, with an individual search costing 0.3 watt-hours of energy [12] while a single circa 2009 Amazon data center operated at 15 megawatts [14]. The EPA estimated in 2007 that data centers consumed 61 terawatt-hours of energy in 2006, 1.5% of the total U.S. consumption, that data centers were the fastest growing electricity consumers, and that an additional 10 power plants would be required for this growth by 2011 [33]. Research studies and reports dating back to the early 1990s (e.g., [25]) have consistently shown low server utilization, with recent reports placing it in the 10–30% ballpark (e.g., [3]). This low utilization feeds into the very public "cloud factories" charge [11] that clouds are bad for the environment and unsustainable. The proposed work hopes to address these issues by *incorporating the individual user*.

Numerous approaches to making data centers more energy efficient have been proposed. The example approach of dynamically choosing the number of servers that are powered up is that has been under investigation for some time. AutoScale [9] is arguably the state of the art here, and the AutoScale paper has a detailed survey of prior work. Given an offered workload, an SLA, and the time needed to bring up/down a server, AutoScale dynamically chooses to bring up or down servers with minimal headroom (additional active servers) and minimal chance of violating the SLAs. Other examples of adapting, in an energy efficient manner, to the offered workload include dynamic voltage and frequency scaling (DVFS) for servers [6], coordinated decisions across the data center [26, 8], and consolidation within the datacenter [34]. The proposed work is likely to be applicable to these and other approaches in that it offers the orthogonal capability of *changing the offered workload.*

SleepServer [1] is a proxy architecture that allows a host to enter low power sleep modes more frequently and opportunistically. A SleepServer machine maintains the network presence of a host and wakes it only when required. Another work [27] simplifies the overall architecture by using a client-side agent. Such proxies are a potential venue for user-centric traffic shaping.

Traffic shaping has had its greatest success in computer networks. It originated in ATM networks [15, 28] and then expanded widely [10, 7], for example into DiffServe [4], and today is widely deployed. The proposed 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.*

3 Frontend augmentation

Our user study is based on popular Android Java applications that are available only in object code form, generally from the Google 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 testcases loaded onto the phone.
2. Continuous measurement of the phone's environment. This includes CPU load, network characterization (RSSI), which radio is in use, ambient light, screen backlight setting, and others.
3. An interface by which the user can supply feedback about performance. A user can supply feedback at any time, but is also can be prompted to do so by a testcase.

The specific choice of applications, testcases, arrival process for testcases, and users is the basis of a study.

Our application augmentation framework is based on Dpartner [37] and DelayDroid [5]. Dpartner is a general framework for decomposing a compiled Java application into its constituent parts, adding interposition, instrumentation, and other elements, repartioning 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.

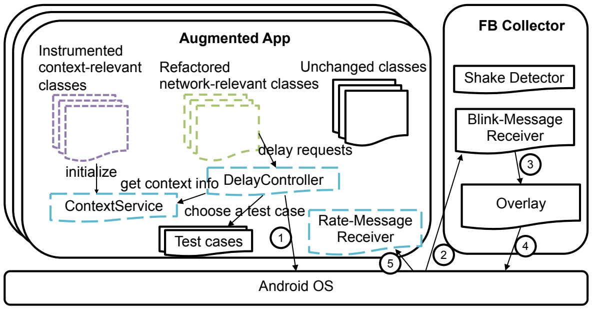

Figure 1: Run-time architecture on the frontend.

DelayDroid leverages Dpartner to augment mobile Android Java applications with support for communication delays and batching, for example to allow study of policies to make 3G/4G radio operation more efficient by reducing tail times. We use DelayDroid specifically to gain the delay capability. DelayDroid effectively introduces an proxy into the application through which high-level (e.g. HTTPWebkit) and low-level (e.g. TCPSocket) network requests pass. The framework interposes the delaying proxy on 110 networking-related functions from five categories within the Android API: HTTPWebkit, HTTPApache, HTTPJavaNet, TCPSocket, and UDPSocket. The proxy can delay requests through all of these interfaces according to our testcase. If no delay is to be introduced (for example, no testcase is being run or the current testcase indicates zero delay), the overhead of the proxy is negligible. The application binary DelayDroid produces 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 (FB Collector) that is responsible for coordinating among augmented applications on the phone, and collecting user feedback.

Figure 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 are those that send network requests, which we interpose on.
- Refactored context-relevant classes are those that we interpose on to track context.
- DelayDroid run-time classes are those that we add in order to inject delay to network requests at run-time.
- Unchanged classes.

The DelayDroid run-time consists of ContextService, DelayController and Rate-Message Receiver. Con-

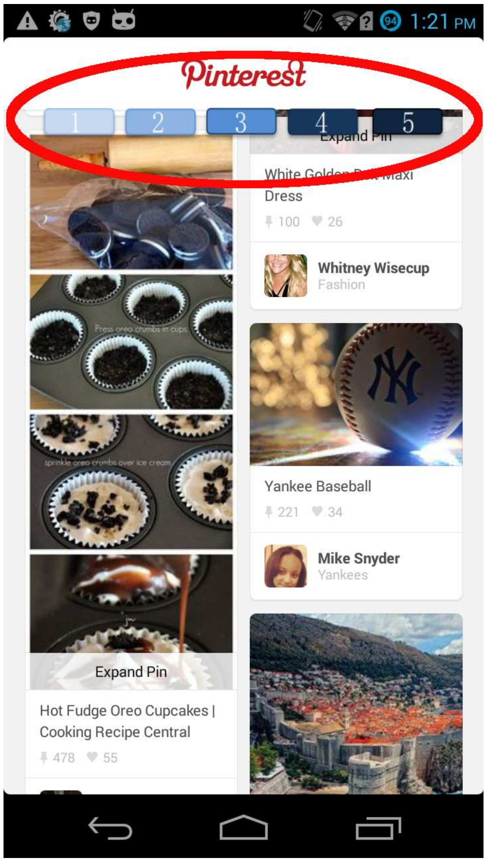
1:21 PM
Pinterest
1
2
3
4
5
Expand Pin
White Golden Dot Maxi Dress
100 26
Whitney Wisecup
Fashion
Press oreo crumbs in cups
sprinkle oreo crumbs over ice cream
Yankee Baseball
221 34
Mike Snyder
Yankees
Expand Pin
Hot Fudge Oreo Cupcakes | Cooking Recipe Central
478 55

Figure 2: Rating overlay in its active state, as hovering over Pinterest.

textService collects and provides information about the context, such as the network status, screen brightness, and CPU usage. DelayController is in charge of introducing additional delays into the network requests. DelayController chooses testcases randomly from within a global pool. Operationally, when any network send request occurs, control flow is detoured through DelayController. DelayController then delays the request according to the testcase. Rate-Message Receiver interacts with FB Collector, and its operation is explained below.

FB Collector is a separate application that collaborates with augmented applications by sending and receiving Android broadcast messages. This is also the user-visible portion of the system. A Likert scale overlay, as seen in Figure 2, hovers over the application. The user can at any time select a rating. 1 indicates complete dissatisfaction with performance, and 5 indicates complete satisfaction with it. Additionally, a shake detector is included, allowing the user to indicate dissatisfaction with device performance by shaking it. In addition to unprompted feedback, the user interface also supports prompted feedback. When we want to prompt feedback, we make the interface blink.

In typical operation as in our user study, DelayController at random times will select a random testcase. The testcase will indicate that network requests should be delayed by a given amount over a given interval. This testcase applies to all augmented applications. DelayController will then apply this delay to all requests that the refactored network-relevant classes route to it. When the testcase is over, DelayController sends a message to FBCollector ①, which is received by the Blink Message Receiver ②, which causes the overlay to blink to prompt the user ③. When feedback is received, FB Collector notifies ④ the Rate-Message Receiver component of the augmented application ⑤ which in turn logs the feedback and all information about the testcase and context.

The log files produced by the system include the following information:

- Timestamp.
- User feedback (via the floating interface), if any, and whether feedback is prompted or unprompted.
- Shake detection status.
- Testcase, if any.
- Arrival time and duration of each network request.
- Foreground application.
- Ambient light level.
- Screen state (on/off, and brightness).
- OS metrics including CPU utilization and load, the percentage of memory free, frequency of each core, and battery percentage.
- Network metrics including type of network (WiFi or cellular), signal strength, and the numbers of packets and bytes sent, received, and dropped.

4 User study

The goal of our user study is to understand the consequences for user satisfaction with performance of delaying network requests from the frontend to the backend. 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 to give us access to popular applications. We then designed a study in which

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: Applications in our study.

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. Figure 3 enumerates the chosen applications, with average request rates having been calculated from traces collected during the study.

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 has 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 in a questionnaire about their familiarity with phones, specific applications, etc. Each was given a $40 gift certificate at the end of their participation.

Our 30 subjects have the following demographics:

- 12 females, 18 males.
- 25 were age 17–25 age, 5 were age 25–35 age range.
- 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 had used one for 2 or more years.
- MapQuest: 7 indicated a familiarity of 7 or greater (on a scale from 1 to 10).
- 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.

Testcases We designed testcases—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. Although we used three kinds of testcases (varying delay, sudden delay, and uniform delay), we use only uniform delay in this paper. In a uniform delay testcase, each network request that occurs during a testcase is delayed by a fixed amount. Our testcases 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 testcase (30 seconds in). Testcases 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 testcase was running was being prompted, but the subject was also so prompted during the zero delay testcase. Neither the subject nor the proctor were aware of which testcase is run and when until after the subject completed the study.

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.

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 on their device.

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.[2] 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

[1] Northwestern IRB Project Number STU00093881.

[2] Study materials will be made available in the final paper.

	Average	Average Change	p-value for Comparison
Delay [ms]	Satisfaction	in Satisfaction	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 4: 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).

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, download 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 of Figure 2, changes to their normal experience of the applications was intentionally kept to a minimum.

5 Study results

Our study produced usable data from 27 of our 30 subjects. Recall that the form of the logged data (Section 3) includes both unlabeled and labeled data, where the labels are the satisfaction the user indicates via the interface. The labels may be prompted (during the middle of a testcase) or unprompted. In the following, we consider the prompted, labeled data only. That is, we consider only user satisfaction during randomly selected one minute windows of time during which we intervene by delaying all network requests by a given amount, including zero. 30 seconds into each window, we prompt the user for a rating from 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 testcase) 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.

5.1 Users tolerate significant additional delay

If we look across all of our users and applications, we see the results of Figure 4. Here we record the average satisfaction for each level of delay, the average change in satisfaction compared to that of the zero delay level, and a p-value. In aggregate, the data seems to be telling us that it is possible to introduce 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), 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 0..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%.

5.2 User 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

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
Google Translate	0.576	0.217	0.646	**0.000**

Figure 5: 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
Google Translate	0.155	**0.003**	0.292	**0.000**

Figure 6: 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.

to the no-delay case. These results are shown in Figures 5 (threshold of 0.5) and 6 (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, 1 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 see 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.

5.3 User tolerance for additional delay varies across users

Figure 7 shows variance of user-perceived satisfaction for each of the delay levels. Recall that our testcases 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, and also the same user differently across time.

Figure 8 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. The number of users varies because we only consider those users for whom there are at least 3 data points represented in their variance calculation.

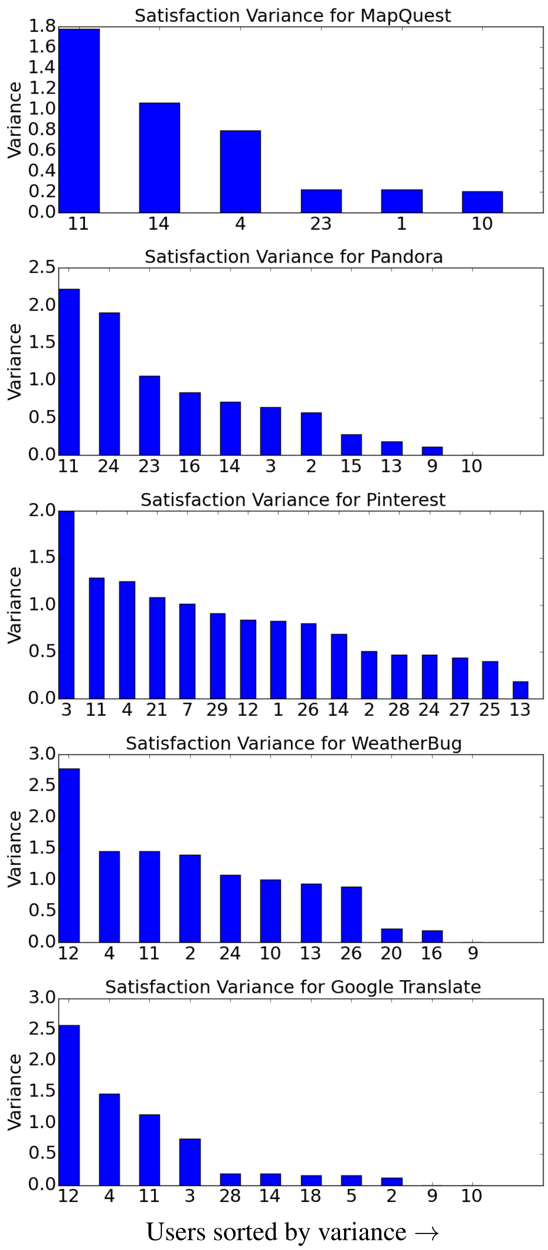

Figure 8: Variance of satisfaction for each user (horizontal axis), ranked by variance.

Since the tolerance for additional delay varies across

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
Google Translate	0.14	1.69	0.54	1.86	0.07

Figure 7: User satisfaction varies considerably across users. Rating variance across users for each combination of application and delay.

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

```
**procedure** PI($t$)
    **return** $K_p \times err(t) + K_i \times \int_0^t err(\tau)d\tau$
**end procedure**
```

Figure 10: PI controller

6 User traffic shaping

We now consider shaping the user traffic by introducing delays. We simulate a system that attempts to limit the arrival rate (keeping load < 1) and to shape the arrivals at any rate to have exponentially distributed interarrival times. Conceptually, our system is a middlebox interposed between the frontend and the backend of the application. The box accepts *user arrivals* of network requests from the mobile application frontend(s), modulates these arrivals by selectively delaying them, and then produces *system arrivals* for the backend.

Our simulation is based on the logs produced by our study, which in addition to user satisfaction information, also include the arrival time of each request. Our logs contain tuples of the form:

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

Recall that each testcase ran for a minute, and during that time delayed *all* network requests. For this reason, we have considerably more requests than testcases. Our simulations make use of 140,401 records of the above form. We can take a stream of requests of this form, whether from a single user or aggregates of users, and apply delays to each in turn to achieve the system's goals. Given the delay we introduced to a request, we can then use the results of Section 4 to estimate the likely impact of the introduced delay on user satisfaction. The general rule we apply here is to keep the additional delays below the 750ms limit we found previously.

6.1 Algorithm

Given a desired system arrival rate, we attempt to delay each request coming in from the user such that the resulting interarrivals of the shaped traffic will appear to be more like those drawn from an exponential distribution, while also obeying the system arrival rate. Such Poisson arrivals are a desirable property both in terms of enhancing the analyzability of systems via queuing theory, and because they are less prone to the Noah Effect [35] in which traffic bursts aggregate across multiple timescales.

We think of our system in terms of load, by which we mean the relationship of the arrival rate of the user arrivals and the arrival rate of the system arrivals, as desired by the backend. The load is the ratio of these two. When the load is less than one, the dominant effect of our system is shaping arrivals to be more Poisson, while when the load is greater than one, the dominant effect is to limit the arrival rate at the backend.

Our algorithm is shown in pseudocode in Figure 9. The system uses EWMA filters to estimate both the current user arrival rate and the current system arrival rate it is providing to the backend. Global feedback is used to compare this current system arrival rate to the desired backend system arrival rate. This error is then fed to a proportional-integrative (PI) controller (Figure 10) that in turn computes a correction. This correction is in the form of the mean of an exponential distribution, the control variable of this system. We then draw from this exponential distribution and use the result as a delay for the current user arrival with respect to the previous system arrival we produced. That is, the system arrivals at the output of our box operate in terms of the controlled exponential distribution. However, these outputs are also constrained by the user arrivals at the input—we never emit a system arrival before its user arrival shows up.

```
procedure SHAPE($u, load, \alpha$)
    $u \leftarrow$ user arrivals
    $inter_u \leftarrow mean(diff(u))$    ▷ compute user interarrivals
    $inter_{desired} \leftarrow inter_u \times load$    ▷ compute desired interarrivals for given load
    $inter_{out} \leftarrow inter_u$
    $s_1 \leftarrow u_1$
    for $i$ in $u$ do    ▷ iterate over arrivals
        $inter_u \leftarrow \alpha \times inter_u \alpha + (1-\alpha) \times (u_{i+1} - u_i)$    ▷ update user interarrival estimate
        $m \leftarrow max(inter_u, inter_{desired})$    ▷ limit to server-selected interarrival
        $err \leftarrow inter_{desired} - inter_{out}$    ▷ estimate error between server-selected interarrival and system interarrival
        $f \leftarrow PI(err)$    ▷ estimate correction via PI controller
        $m \leftarrow max(0.0001, m + f)$
        $delay \leftarrow exprnd(m)$    ▷ generate corrective exponential delay
        $s_{i+1} \leftarrow s_i + delay$    ▷ delay user arrival with corrective exponential delay
        if $u_{i+1} > s_{i+1}$ then    ▷ but only if possible (system arrival must occur after user arrival)
            $s_{i+1} \leftarrow u_{i+1}$
        end if
        $inter_{out} \leftarrow \alpha \times inter_{out} + (1-\alpha) \times (s_{i+1} - s_i)$    ▷ update system interarrival estimate
    end for
end procedure
```

Figure 9: Traffic shaping algorithm

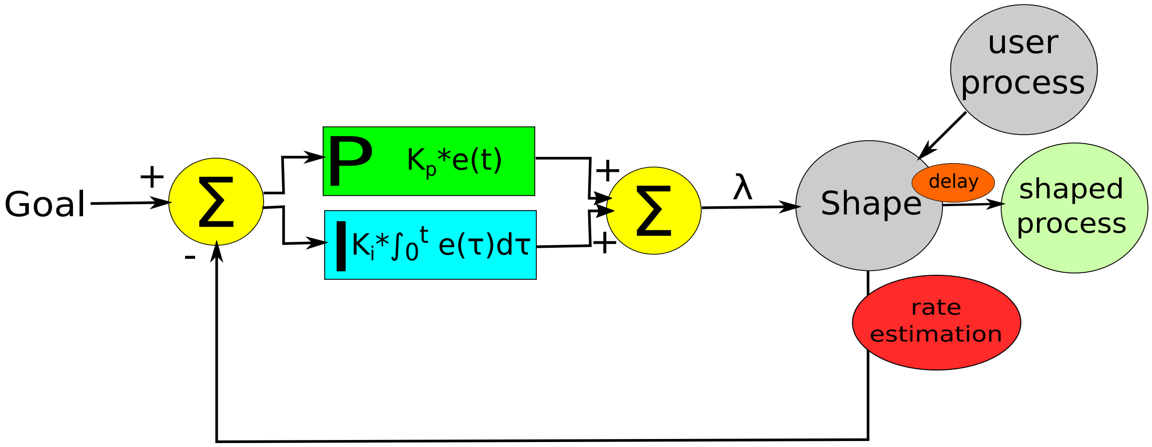

Figure 11: Traffic shaping system used to simulate the effect of shaping user traces with our algorithm. PI controller biases the mean of the exponential used to delay requests, creating the shaped process.

6.2 Evaluation

We simulated the proposed system separately on each user trace collected during the user study. For each user, the trace was segmented into "sessions", where a session was defined to be any section of network requests—that is, application activity—where no interarrival was longer than 10 seconds. The algorithm was applied to each session, resulting in a shaped user trace. We consider the effect of different system arrival rates by varying the load, allowing us to compare across users. Note that our algorithm itself is driven by a random process (the exponential random number generator), as is our evaluation of how close our output is to having exponential interarrivals. For each trace, we execute the algorithm 100s of times, with different seeds, to attempt to estimate the ensemble average behavior, which is what we report. The entire simulation system is shown in Figure 11

In all cases, we are able to achieve the rate-limiting goal of our system, and, as expected, delays introduced grow as we approach a load of 1.0. We also want to evaluate "how much more Poisson" our system's output is. To do so, we consider quantile-quantile plots (QQ plots) comparing our system's input (the user arrivals) and output (the system arrivals) with actual Poisson arrivals. Figure 12 shows an example. The desired shape of data on these QQ plots is that of the dotted line. If all data points followed the dotted line, we would consider the distributions matched. Notice that the shaped system arrivals (right graph) are closer to this than the unshaped user arrivals (left graph). We fit a line to each graph and quantify this fit with its R^2 value. We can then compare the unshaped and shaped arrivals by the difference in their R^2 values, which we denote ΔR^2. Here, the unshaped user arrivals have $R^2 = 0.5982$, the shaped system arrivals have $R^2 = 0.8319$, and thus $\Delta R^2 = 0.2337$. Recall that R^2 ranges from -1 to 1, so this example shows shaped system arrivals that are considerably more Poisson than their corresponding unshaped user arrivals.

As system load increases, we have more opportunity to slow down requests, so we would expect that ΔR^2 will grow with increasing load. On the other hand, increasing load will also mean we will need to introduce larger amounts of delay, as well as more opportunities for one request's delay to bleed into the next's, compounding its delay. Hence, the system will need to trade off between ΔR^2 (how much more Poisson the system arrivals are than the user arrivals) and the delays the user sees. If we increase delay too much, user satisfaction will drop.

Figure 13 shows ΔR^2 and the added delay across all of our traces. Delay is considered both in terms of the average, and the 90th and 95th percentiles. Recall that our user study suggested that most users are largely unperturbed by delays up to 750ms. We see that our shaping algorithm is able to introduce appropriate delays for a load of 0.4 while keeping the introduced delay below this threshold for 95% of requests. A load of 0.5 can be supported while keeping delay within this bound for 90% of requests. The R^2 fit is improved by 0.01948 and 0.03440, respectively.

Load	ΔR^2	Avg Delay	95 %ile	90 %ile
0.1	0.00053	0.00355	0.00000	0.00000
0.2	0.00063	0.01273	0.00000	0.00000
0.3	0.01409	0.11179	0.00000	0.00000
0.4	0.01948	0.37374	1.61987	0.00000
0.5	0.03440	0.79270	4.87666	0.69274
0.6	0.05372	1.50162	9.11135	4.07337
0.7	0.08436	2.63141	18.22722	8.58257
0.8	0.10060	5.16139	28.00491	14.96098
0.9	0.12190	9.57013	38.44012	30.49053
1.0	0.14322	18.72321	53.37096	48.03965

Figure 13: Ability to make traffic more Poisson increases with necessity to introduce delay, $K_p = 0.9, K_i = 0.5, \alpha = 0.8$. Within the 750ms bound of Section 4 a load of 0.4 to 0.5 can be supported while keeping delays low enough that they do not change user satisfaction with performance.

7 Conclusions

Conventional approaches to configuring datacenter backends of mobile applications rely on the assumption that the workload generated by the frontend of the application being sacrosanct, and responding to changes primarily at the datacenter. Performance requirements are assumed to be uniform across all users, and once again the datacenter is tasked to respond to generated workloads and satisfy them for this uniform performance. In this work, we consider that users may have both varied expectations and tolerances for the performance of request fulfillment, and that we can use this to our advantage to shape traffic at the user/frontend level, in order to provide a more beneficial workload at the datacenter.

We conducted an "in the wild" user-study, which provided users with augmented versions of 5 popular Android applications, that would introduce delays into the network requests being made by those applications, and query the subject for their satisfaction with the performance of the application. We found that on average, for all of our test applications, subjects were not adversely affected by introduced delays of up to 750ms. When an-

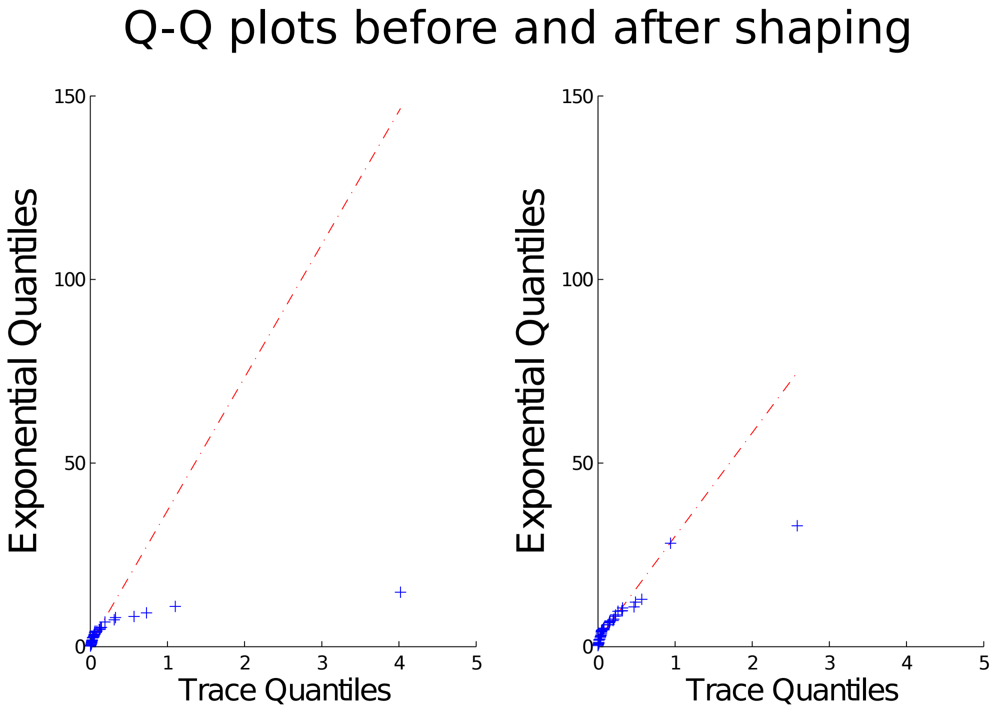

Figure 12: Example QQ plot for user 1, comparing interarrivals of the user arrival process (left) and the interarrivals of the system arrival process (right). The shaping algorithm improved exponential fit by $\Delta R^2 = 0.2337$.

alyzing individual applications, we found that certain applications were more sensitive to delays, suggesting an opportunity for a class of applications to have their requests delayed, making the workload at those datacenters more beneficial. By analyzing user reactions, we noticed that certain users were much more sensitive to introduced delay, which both bolsters our original thought that performance requirements needn't be uniform across users, and presents an opportunity to delay some users more than others, which would lessen an aggregate workload while affecting users satisfaction less overall.

We developed a traffic shaping algorithm that would introduce exponential delays to network requests, and simulated the effects of applying it by shaping user request traces that we had collected during the user study. We found that we could skew the interarrival process of user traces to look more like a Poisson process by a factor of $\Delta R^2 = 0.01948$ while delaying 95% of requests within user tolerance, and by $\Delta R^2 = 0.03440$ while delaying 90% of requests within user tolerance.

Future Work In this paper we have identified the opportunity for shaping the workload generated by end-users using mobile applications. We would like to extend our user study to study more applications, more users, and collect more data in order to gain even better insights into how users react to introduced delays. With more user data, we would like to build user models to evaluate how our traffic shaping is likely to affect each specific user, and leverage individual variations in tolerance to introduce as much delay as possible to each user without detrimentally affecting their satisfaction.

We would also like to simulate the effect of the aggregate of shaped user workloads at the datacenter—our idea is that by shaping user traffic to appear more like a Poisson process, the aggregate user traffic will both be more predictable and have a lower rate. We would like to study how real datacenter load balancers react to shaped traffic vs real life collected traces.

References

[1] Agarwal, Y., Savage, S., and Gupta, R. Sleepserver: A software-only approach for reducing the energy consumption of pcs within enterprise environments. In Barham and Roscoe [2].

[2] BARHAM, P., AND ROSCOE, T., Eds. *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010* (2010), USENIX Association.

[3] BARROSO, L., AND HOELZLE, U. The case for energy-proportionate computing. *IEEE Computer 40*, 12 (December 2007), 33–37.

[4] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An architecture for differentiated services. Tech. Rep. RFC 2475, Network Working Group, December 1998.

[5] CAI, H., ZHANG, Y., SWIECH, M., HUANG, G., AND DINDA, P. Delaydroid: An automatic framework for reducing tail time energy in existing, unmodified android applications. In *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (May 2015). In Submission.

[6] DENG, Q., MEISNER, D., BHATTACHARJEE, A., WENISCH, T., AND BIANCHINI, R. 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).

[7] ELWALID, A., AND MITRA, D. 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).

[8] GANDHI, A., HARCHOL-BALTER, M., DAS, R., AND LEFURGY, C. 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).

[9] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems 30*, 4 (November 2012), 1–26.

[10] GEORGIADIS, L., GUERIN, R., PERIS, V., AND SIVARAJAN, K. Efficient network qos provisioning based n per-node traffic shaping. *IEEE/ACM Transactions on Networking 4*, 4 (1996), 482–501.

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

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

[13] GUPTA, A., LIN, B., AND DINDA, P. A. 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).

[14] HAMILTON, J. Where does the power go in high scale data centers? Keynote of ACM SIGMETRICS 2009.

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

[16] LANGE, J., DINDA, P., AND ROSSOFF, S. Experiences with client-based speculative remote display. In *Proceedings of the USENIX Annual Technical Conference (USENIX)* (2008).

[17] LANGE, J. R., MILLER, J. S., AND DINDA, P. A. Emnet: Satisfying the individual user through empathic home networks. In *Proceedings of the 29th IEEE Conference on Computer Communications (INFOCOM)* (March 2010).

[18] LIN, B., AND DINDA, P. User-driven scheduling of interactive virtual machines. In *Proceedings of the Fifth International Workshop on Grid Computing* (November 2004).

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

[20] LIN, B., AND DINDA, P. Towards scheduling virtual machines based on direct user input. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (2006).

[21] LIN, B., MALLIK, A., DINDA, P., MEMIK, G., AND DICK, R. 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).

[22] MALLIK, A., COSGROVE, J., DICK, R., MEMIK, G., AND DINDA, P. 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).

[23] MALLIK, A., LIN, B., MEMIK, G., DINDA, P., AND MEMIK, G. User-driven frequency scaling. *IEEE Computer Architecture Letters 5*, 2 (2006).

[24] MILLER, J. S., MONDAL, A., POTHARAJU, R., DINDA, P. A., AND KUZMANOVIC, A. 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.

[25] MUTKA, M. W., AND LIVNY, M. The available capacity of a privately owned workstation environment. *Performance Evaluation 12*, 4 (July 1991), 269–284.

[26] NATHUJI, R., AND SCHWAN, K. Virtualpower: Coordinated power management in virtualized enterprise systems. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP 2007)* (October 2007).

[27] REICH, J., GORACZKO, M., AND KANSAL, A. Sleepless in seattle no longer. In Barham and Roscoe [2].

[28] REXFORD, J., BONOMI, F., GREENBERG, A., AND WONG, A. Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches. *IEEE Journal on Selected Areas in Communications 15*, 5 (1997), 938–950.

[29] SHYE, A., B. OZISIKYILMAZ, A. M., MEMIK, G., DINDA, P., DICK, R., AND CHOUDHARY, A. 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).

[30] SWIECH, M., AND DINDA, P. 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).

[31] TARZIA, S., DICK, R., DINDA, P., AND MEMIK, G. 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.

[32] TARZIA, S., DINDA, P., DICK, R., AND MEMIK, G. Display power management policies in practice. In *Proceedings of the 7th IEEE International Conference on Autonomic Computing (ICAC 2010).* (June 2010).

[33] UNITED STATES ENVIRONMENTAL PROTECTION AGENCY. Report to congress on server and data center energy efficiency public law 109-431, August 2007.

[34] VERMA, A., AND DASGUPTA, G. Server workload analysis for power minimization using consolidation. In *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009* (2009), G. M. Voelker and A. Wolman, Eds., USENIX Association.

[35] WILLINGER, W., TAQQU, M. S., SHERMAN, R., AND WILSON, D. V. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. In *Proceedings of ACM SIGCOMM '95* (1995), pp. 100–113.

[36] YANG, L., DICK, R., MEMIK, G., AND DINDA, P. Happe: Human and application driven frequency scaling for processor power efficiency. *IEEE Transactions on Mobile Computing (TMC) 12*, 8 (August 2013), 1546–1557. Selected as a Spotlight Paper.

[37] ZHANG, Y., HUANG, G., LIU, X., ZHANG, W., MEI, H., AND YANG, S. Refactoring android java code for on-demand computational offloading. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA 2012)* (October 2012).