Give in to Procrastination and Stop Prefetching

Lenin Ravindranath
Microsoft Research & M.I.T.
Redmond, WA, USA
lenin@csail.mit.edu

Sharad Agarwal
Microsoft Research
Redmond, WA, USA
sagarwal@microsoft.com

Jitendra Padhye
Microsoft Research
Redmond, WA, USA
padhye@microsoft.com

Christopher Riederer
Columbia University
New York, NY, USA
cjr2149@columbia.edu

Abstract – Generations of computer programmers are taught to prefetch network objects in computer science classes. In practice, prefetching can be harmful to the user's wallet when she is on a limited or pay-per-byte cellular data plan. Many popular, professionally-written smartphone apps today prefetch large amounts of network data that the typical user may never use. We present Procrastinator, which automatically decides when to fetch each network object that an app requests. This decision is made based on whether the user is on Wi-Fi or cellular, how many bytes are remaining on the user's data plan, and whether the object is needed at the present time. Procrastinator does not require developer effort, nor app source code, nor OS changes – it modifies the app binary to trap specific system calls and inject custom code. Our system can achieve as little as no savings to 4X savings in bytes transferred, depending on the user and the app. In theory, we can achieve 17X savings, but we need to overcome additional technical challenges.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design — Network Communications

General Terms

Design, Measurement, Performance

1. INTRODUCTION

Many smartphone apps rely on network connectivity for key app functionality. To mitigate the impact of cellular network delays on app performance, app developers often resort to *prefetching*. They download network content before the user needs it - for example to populate images that are off-screen. A typical example is shown in Figure 1. This popular weather app downloads a large amount of data as soon as the app is launched, including many images. While some of the data is necessary to display current weather on the "main" page of the app, many prefetched images are displayed only deep within the app. Our intuition, buttressed by an informal user study (§4), suggests that many of these prefetched images are rarely accessed.

Hotnets '13, November 21–22, 2013, College Park, MD, USA.

Those users that scroll or click through an app to visit that off-screen content will experience a more responsive app. However, for those users that do not visit that off-screen content, the penalty is wasted network consumption. This waste can harm two sets of users – (1) users that are always conscious about data consumption because they are on a pay-as-you-go plan or limited data bytes plan; (2) users that start their monthly cellular billing cycle with a large number of bytes (e.g. 2GB), but eventually run low and want the remaining bytes to last them through the end of their cycle.

This problem cannot be solved by producing two versions of an app – data-light and data-heavy versions – for two different sets of users. A data-rich user can sometimes become data-poor, or vice-versa temporarily when the user connects to Wi-Fi (we assume that most Wi-Fi connectivity is free or significantly cheaper per byte) or starts running out of her monthly cellular allotment. Alternatively, producing a single adaptive app is difficult because it would have to manage multiple network transfers that affect different parts of the user interface. Worse, the user can be scrolling around in the app while some network transfers are ongoing and connectivity changes between cellular and Wi-Fi.

We have embarked on building a system to solve this problem. Our goal is to *automatically* delay prefetching of network content *when appropriate*, to reduce data usage. The system is (aptly) named *Procrastinator*. The "delaying" or "procrastination" is done based on current network connectivity (cellular or Wi-Fi), the status of the user's cellular data plan (such as plentiful or in danger of exceeding), where in the app the user is (such as which parts of the UI are visible), and impact on other functionality of the app beyond the UI (such as playing music or vibrating the phone).

In designing Procrastinator, we strive for *"immediate deployability"* – we do not require changes to the mobile OS, nor runtime. We also attempt to achieve *"zero effort"* for the app developer – we do not require her to write additional code, nor add code annotations. Procrastination is done by automatically rewriting app binaries. We also strive for *"zero functionality impact"* – beyond appearing as though the network is occasionally slower, the user should not experience any other change in the functionality of the app, as seen in our demo [3].

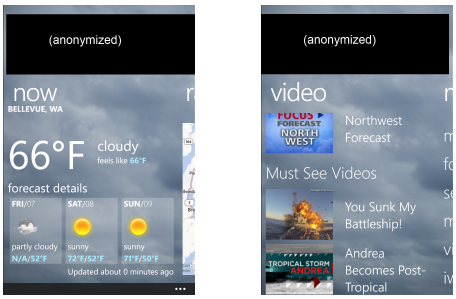

Figure 1: Screenshots of a popular weather app. The first screen is visible immediately after app launch. The second screen is visible only after two swipes.

We have implemented a preliminary version of Procrastinator for the Windows Phone 8 platform. There are two key challenges in building Procrastinator. First, Procrastinator must automatically identify asynchronous network calls that are candidates for procrastination. This involves careful static analysis of the app code (§3.1). Next, Procrastinator must re-write the app code, so that at run time (§3.2), it can decide (a) whether to procrastinate each candidate call, and (b) when to execute a previously procrastinated call.

We evaluate our prototype using lab experiments and a small study of 9 users with 6 popular Windows Phone apps. Since Procrastinator does not need app source code, we use third party app binaries. We find that even this simple prototype can reduce app network usage by as much as a factor of 4. Informal exit interviews with users suggest that none of them experienced the loss or change of app functionality beyond occasional slowdown in loading of images.

While we have built and evaluated our prototype, much work remains to be done to build a robust practical system. We describe these open issues in §5.

2. PROBLEM DEFINITION

We analyzed the binary code of a large set of Silverlight apps in the Windows Phone app store to understand programming patterns used by developers to prefetch content. While diverse, the programming patterns used tend to fall into three categories. A representative example of each category is shown in Figure 2. Even though we analyzed .NET byte code, for convenience, we show pseudo-code.

Before we describe these examples, we highlight two relevant aspects of the Windows Phone programming framework. First, the framework supports only asynchronous network I/O. Second, images and text are displayed on the screen by assigning them to special UI widgets.

In Pattern 1, the developer assigns an image to an image widget that is not yet visible on the screen. One app that uses this pattern is a news reader app. Each news story has an associated image. The news stories are displayed as an "infinitely scrolling" list. When the app is launched, only the top three or four stories and images are visible to the user. However, the app continues to fetch images that are "below the fold" in anticipation of user scrolling down. Note that the HTTP fetch call executes asynchronously and the image is populated after the call successfully returns.

```
Pattern 1
void page_load() {
  /* imageWidget not visible on the current screen */
  imageWidget.Source = http.fetchImage(url);  }

Pattern 2
void page_load() {
  http.FetchData(url, callback);  }
void callback(result) {
  var cleanText = clean(result);
  /* textWidget not visible on the current screen */
  textWidget.Content = cleanText;  }

Pattern 3
void mainpage_load() {
  http.FetchData(url, callback);  }
void callback(result) {
  hurricaneWarnings = parseXML(result);  }
/* called when user clicks to navigate to a new page */
void navigate_hurricaneWarningsPage() {
  hurricaneWidget.Content = hurricaneWarnings;  }
```

Figure 2: Common prefetching patterns in apps.

Programmers often explicitly account for these asynchronous fetches, especially if additional processing is required before displaying the fetched data. This is illustrated in Pattern 2. The fetched text is "cleaned up"and then displayed by assigning it to a text widget box. The text widget box may not yet be visible on the screen, but once the user scrolls down to it, it will have the text ready for viewing.

Pattern 3 is similar to Pattern 2, but the assignment to the UI widget is delayed even further. This pattern is used in a weather app. At launch, the app fetches data about hurricane warnings, which it stores in a global variable. The data is used only if the user navigates to a specific tab (or a "page") within the app that displays hurricane warnings.

Our goal is a system that detects such patterns, and delays network calls until the object in question is actually needed. We can achieve this by automatically rewriting app code. For example, in Pattern 1, we can insert code such that the network call is made only when the image widget is visible.

3. SYSTEM DESIGN

Figure 3 shows an overview of Procrastinator. It has two main components: the Instrumenter and the Procrastinator Runtime. The Instrumenter can take any app binary and produce a *procrastinated* version. It statically analyzes the app to look for prefetching patterns and finds candidates for procrastination. It then rewrites the associated binary code and incorporates the Procrastinator Runtime into the app code.

When the app is run, the Procrastinator Runtime dynamically checks the state of the network, tracks the UI, and delays those network calls whose results are not immediately needed by the UI. These procrastinated network calls are then fetched on demand when the user navigates to those portions of the UI. It delays calls only when the user is on cellular and low on data bytes.

Since the Instrumenter does not require developer support

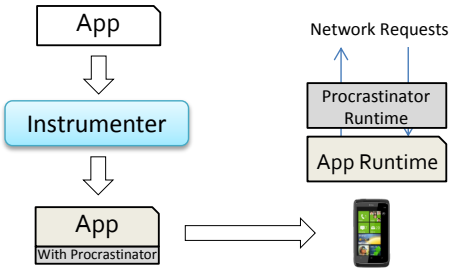

Figure 3: Procrastinator design overview.

nor app source code, it can be run as part of the app store, the OS platform or as a third-party service.

3.1 Instrumenter

The Instrumenter statically analyzes each network call in the app, to identify network calls that can be procrastinated. It rewrites these calls to make them go through the Procrastinator Runtime as shown in Figure 4. We later describe how the runtime handles these calls, but first we describe why the instrumentation process itself is non-trivial.

For network requests that directly update UI widgets, in addition to passing the parameters of the network call (such as the URL), the rewritten call also passes along references to corresponding UI widgets set by the response. Instrumenting Pattern 1 is relatively straightforward, since only a single UI widget is updated by the network call, and the updated widget is evident in the code.

Instrumenting Pattern 2 is more challenging. First, statically identifying the UI widgets that are updated is not trivial. In the example shown, the callback method itself updates the UI widget. However, the callback method could call other methods (perhaps asynchronously!) and the updated UI widgets may lie deep in this call tree. To address this problem, we statically construct a *conservative* call graph (one that includes all possible code paths) that is rooted in the callback method, and analyze all code in it to discover all UI widgets being updated. Second, the callback method or any code in its call graph may have *side effects* besides updating the UI. For example, one of the methods may vibrate the phone or store a current timestamp in a global variable. We analyze all code in the call graph to ensure that it has no side effects. If any code in the call tree has any side effects, the Instrumenter conservatively decides that the call cannot be procrastinated, and the code is not modified.

Procrastinating code like Pattern 3 is challenging as well. Here, the result of the network request is stored in memory and used later. This is may be done to prefetch data for widgets that have not yet been instantiated. As in the previous pattern, we analyze the call graph of the callback and consider it for procrastination only if the there are no side effects other than updating a few global variables. We identify these global variables and instrument both the network request and access to the global variables as shown in Figure 4.

```
Pattern 1:
void page_load() {
  /* imageWidget not visible on the current screen */
  CheckProcrastinate(imageWidget, url);  }

Pattern 2:
void page_load() {
  CheckProcrastinate(url, callback, textWidget);  }
void callback(result) {
  var cleanText = clean(result);
  /* textWidget not visible on the current screen */
  textWidget.Content = cleanText;  }

Pattern 3:
void mainpage_load() {
  /* 2 is a unique id that identifies
  the network request */
  Procrastinate(url, callback, 2);  }
void callback(result) {
  hurricaneWarnings = parseXML(result);  }
/* called when user clicks to navigate to a new page */
void navigate_hurricaneWarningsPage() {
  /* 2 identifies the network request to fetch */
  FetchProcrastinated(2);
  /* waits until the network call is complete */
  wait();
  hurricaneWidget.Content = hurricaneWarnings;  }
```

Figure 4: Rewriting code to add Procrastination.

3.2 Procrastinator Runtime

The Procrastinator Runtime simply executes the given network calls when the user is on Wi-Fi or when there is no cellular data plan pressure. When the phone is on cellular and there is data plan pressure, it delays network calls and executes them when needed. Our runtime uses our Windows Phone 8 Data Sense feature, which tracks overall data consumption against the user's data plan, to determine when the phone is running out of cellular data bytes [1].

For the first two patterns of procrastination shown in Figure 4, the runtime monitors those UI widgets that are attached to the request. If none of them are visible during the request, it delays the network call and adds it to the list of procrastinated requests.

All procrastinated requests are reevaluated every time the UI changes. We rewrite the app to add event handlers to UI manipulation events and UI update events to detect UI changes. When the UI changes, the runtime traverses each of the procrastinated requests and checks if any of the UI widgets associated with the request is visible. If any are visible, it makes the network call and removes it from the list.

For the third pattern in Figure 4, the runtime delays the network call by default during the request and adds it to a list with the unique id passed with the request as the key. The call is delayed until one of the global variables modified by the callback is accessed. We intercept all accesses to the identified global variables and pass the same unique id to inform the Procrastinator Runtime to fetch the request before the global variable is first accessed. Access to the variable is blocked until the procrastinated network call completes.

4. EVALUATION

To test the effectiveness of Procrastinator, we pick 6 apps from the top 50 in the marketplace, listed in Table 1. We are not allowed to publicly disclose the identities of the apps.

app	functionality	action
App1	cooking recipes	read top daily recipe
App2	movie times	see details of top movie
App3	movie times	see details of top movie
App4	news aggregator	read top news story
App5	news paper	read top news story
App6	weather reports	see current weather and forecast

Table 1: The 6 popular Windows Phone 8 apps we evaluate and the action we perform in each run of the app in lab experiments. Each app is authored by the relevant company (e.g. Facebook by Facebook).

Using controlled lab experiments and a small scale user study, we demonstrate how Procrastinator can reduce data usage in these apps. In our evaluation, we use Procrastinator only on prefetching code that uses Pattern 1 (Figure 2). Patterns 2 and 3 often process non-media network objects in complex ways before displaying them to the user. Our current implementation is not yet robust enough to handle all such scenarios. We believe that focusing on images in Pattern 1 should get us the bulk of the savings with less complexity and risk of unintentionally altering app behavior.

4.1 Lab experiments

We installed the 6 original apps from Table 1, as well as their procrastinated versions on a Nokia Lumia 920 phone. The procrastinated apps have different app IDs and hence are completely isolated from their original version – they do not share any state, storage, or web caches. The phone had only Wi-Fi connectivity. All of the phone's traffic was intercepted by a server running the Fiddler [2] proxy.

We ran both versions of each app, performing the action listed in Figure 1, and captured traces of network activity from the Fiddler proxy. We ensured that there was no background activity from other apps and OS features. Normally, Procrastinator would not procrastinate network calls if the user is connected to Wi-Fi. For the purposes of our evaluation, we turned off this feature.

We manually inspected the Fiddler traces and compared the content that was fetched to what was displayed on the screen. We then classified each web object that was fetched by the original app into one of three buckets. A web object is considered "necessary" if the contents were used to show something to the user on the main screen of the app or any of the subsequent screens when performing the action listed in Table 1. In cases where we cannot deduce whether the content of web objects are used in the screen (typically when we cannot decipher the contents), we conservatively assume them to be "necessary". If a web object is not necessary, then it is either "skipped" or "prefetched" – if the Procrastinated version of the same app does not download this object, then it is "skipped", otherwise it is "prefetched". This classification is not subjective. Apart from verifying what appeared on the screen, we also carefully looked through the app's binary source code to determine whether the object was needed to render the page that the user saw.

The results are summarized in Figure 5. For all tested

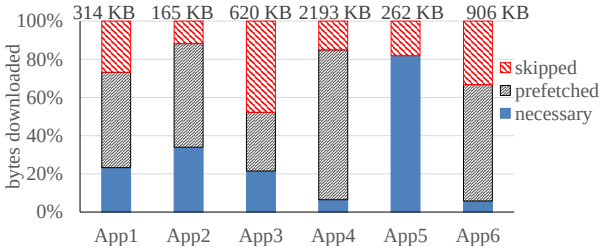

Figure 5: Bytes consumed when running each of the 6 original apps, and their Procrastinated versions.

object	type	bytes
ad	xml	6,226
weather	js	6,096
alerts	xml	54
weather detail	js	6,562
forecast	xml	3,466
weather	js	880
weather	js	640
weather detail	js	5,622
sun rise/set	js	43
storm info	xml	572
weather detail	js	458
ad	text	6,700
ad	js	223
ad	oml	6,546
ad	gif	8,493
ad	gif	1,097
ad	gif	43

object	type	bytes
video list	xml	302,940
photo list	js	107,463
photo status	js	84
regional weather	xml	26,778
photo list	js	11,616
national forecast	xml	4,614
national forecast	xml	12,065
tropical forecast	xml	2,837
weather news	xml	96,810
radar map†	png	280,933
video screenshot†	jpeg	9,965
video screenshot†	jpeg	11,192
video screenshot†	jpeg	2,817
video screenshot†	jpeg	3,681

Table 2: Breakdown of 31 objects fetched by a single run of the unmodified App6. We manually classify the left 17 items as "necessary" objects, and the right 14 items as "prefetched" objects. The objects marked with † are those that Procrastinator did not fetch. The bytes exclude TCP/IP and HTTP headers.

apps, the non-necessary content dominates the number of bytes downloaded. For example, the original App6 downloads 31 web objects, for a total of 906 KB. Our inspection shows that that 52 KB of those were necessary. We briefly describe these objects in Table 2. Of the remaining 853 KB that the original app downloaded, Procrastinator correctly identified 301 KB of images as not necessary and did not download them. However, there was an additional 552 KB of non-image content that our current implementation of Procrastinator does not identify as wasted and allows the app to download them immediately. Hence in this situation, Procrastinator saved 33% of the total bytes, but ideally could have saved as much as 94% of the 906 KB total.

To understand what objects Procrastinator is skipping, we present screenshots of the original App6 in Figure 6. The left picture is the first screen presented to the user. The current weather and the forecast is shown, along with some icons and occasionally ads will be shown to the user on the top of the screen. If the user swipes to the right, she is shown the middle picture, which has a large radar map. If user swipes again to the right, a list of four weather videos is presented with images for each of the videos. If the user remains on the first screen and then quits the app, then the radar map and the video images are all wasted downloads. Procrastinator is able to correctly identify those as wasted because

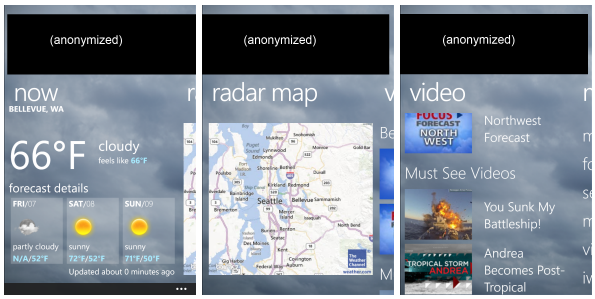

Figure 6: Screenshots of App6.

the screen coordinates for those images are off-screen, and hence does not download those 5 objects listed at the bottom right of Table 2. The app also downloads some large XML files that describe photos that users have taken of weather phenomenon across the US and detailed weather news for other parts of the US. These are shown to the user if the user swipes three times over to the right and clicks on a menu item. Our current implementation of Procrastinator only handles image procrastination, and hence does not handle this situation.

Other apps are similar. App1 downloads 40 objects. Many of these objects are small, and contain text descriptions of recipes. The largest objects are photos of recipes, which Procrastinator automatically targets. App3 downloads 110 web objects (wow!), most of which are small XML blobs describing each movie, review details, news and user comments. Approximately 700 KB of images are downloaded, and Procrastinator targets those. App4 downloads news articles and news images, of which the latter is targeted by Procrastinator. App5 downloads relatively few objects, but does prefetch images, all of which Procrastinator correctly procrastinates.

4.2 User trial

Our lab experiments show that Procrastinator can substantially reduce data consumption. However, the savings experienced by real users in the wild will depend on multiple factors including network conditions, specific user interactions, and app caching. We also did not evaluate in the lab the additional delay experienced by the user when a procrastinated object has to be fetched, such as in the weather app, if the user swipes to see the radar image. To evaluate these aspects, we deployed the procrastinated versions of these 6 apps to 9 colleagues for 6 weeks. We asked the users to use any of these apps that they like as normally as they would if they discovered it in the app marketplace.

All 9 users had unlimited data on their cellular plans. Since the phones are never in danger of exceeding their data limits, the procrastinated apps always fetch all web requests by the apps. However, we also record in a log file when each web request is made, whether it would have been procrastinated, when the transfer finishes, how many bytes it consumed, and when it is displayed to the user (if at all). These

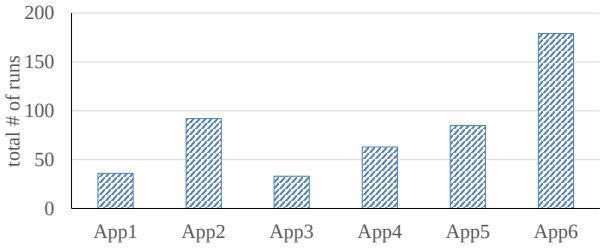

Figure 7: Total number of user sessions per app.

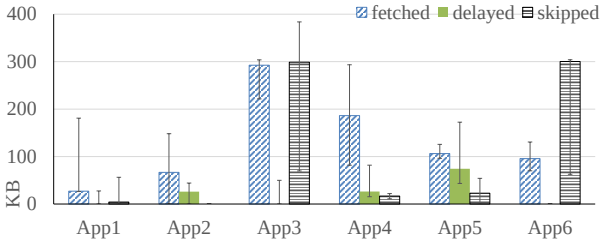

Figure 8: Average KB fetched and saved per session, with 25th and 75th percentiles, across all sessions.

logs are transferred in the background, when over Wi-Fi, to an Azure database in the cloud. In this way, we can calculate the number of bytes that were fetched, skipped, and delayed.

Figure 7 shows the number of app launches across all users for each app. Apparently our users check the weather more frequently than they cook with a new recipe. Figure 8 shows the number of bytes spent on average per user session of each app. The bytes are split into three categories. The "fetched" bytes are those that the app requested where either the object was not an image, or was an image that Procrastinator detected was visible to the user. The "skipped" bytes are for those images that the app fetched, but Procrastinator would have skipped if the user was low on their data plan, and were never shown to the user. The "delayed" bytes are for those images that the app fetched, Procrastinator would have skipped, but then would have to fetch because the user was later shown that image.

We see that in the case of App6, Procrastinator reduces the number of bytes by over 4X on average. In the other extreme, for App2, Procrastinator almost never saves any bytes, perhaps because users scroll through all available movies before leaving the app. We also note that for some apps, such as App3 (which shows the user more photos of each movie), there is huge variance in the number of bytes saved.

We contrast this with Figure 9, where we observe that for App6, a small number of objects are skipped, but those objects contribute a lot to the number of bytes. This matches our original intuition of images consuming a lot of bytes, and hence our current focus on procrastinating those objects. In contrast, App3 has a lot of small images (stills from movies), and procrastinating many of those small images adds up.

In some cases, the user would incur no delay in using a procrastinated app because she does not scroll to a part of the app where a previously skipped image is shown. For other

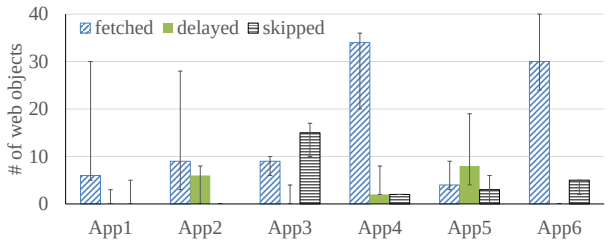

Figure 9: Average number of web objects fetched per session, with 25th and 75th percentiles, across all sessions.

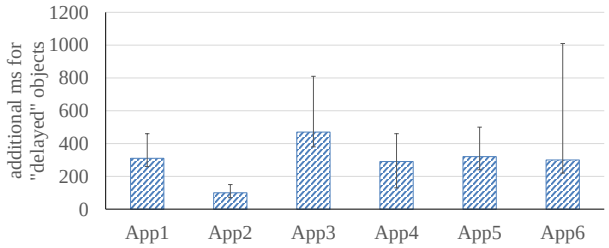

Figure 10: Average additional delay incurred per delayed image, with 25th and 75th percentiles, across those images incorrectly procrastinated in user trial.

cases where the user is shown a skipped image, we calculate the delay that she incurs. This is the smaller value of either the download time for that image, or the difference between when the app originally requests the image and when Procrastinator detects the image is visible. Figure 10 shows this additional delay for only those images that were "delayed" in the previous figures. We find that the additional delay is under 1 second, and typically below 500ms. We believe this is small enough that users will only occasionally notice it, and when the user is running low on their cellular data plan, will be willing to tolerate for the cost savings.

While our set of users is small and may not necessarily be representative of typical users, our findings are at least indicative of the potential savings that users can get from Procrastinator. Today, we can achieve as much as 4X savings for a particular app, to as little as no savings for another app. If we go beyond images to also Procrastinating other web objects, the potential there can be as much as 17X savings.

5. FUTURE WORK

Programming Support: A major limitation of automatic procrastination as we describe in § 3 is that we are forced to be highly conservative. A network request can be procrastinated only if the entire call graph rooted in its callback method is side-effect free in all possible execution paths, except for updating UI widgets or a few global variables. Hence, we are unable to procrastinate many network calls, and hence our focus on images in our evaluation. If app developers cooperate, they could structure their code to allow for more procrastination. To this end, we are considering building a Visual Studio plugin which monitors app code during compile time, and identifies whether each network call is procrastinatable. If necessary, it points to the appropriate portion of the code that has side-effects that are preventing procrastination for each network call. The app developer can then re-factor the code.

Balancing speed and cost: The user experiences a delay when a procrastinated network object has to be fetched, such as in the weather app, if the user swipes to see the radar image. One way to minimize this delay is to predict which objects the user may view, and not procrastinate those prefetches. For each app, the procrastinator runtime could keep a log of which procrastinated network objects had to be eventually fetched, and use this log to "learn" individual user behavior, and then adapt.

6. RELATED WORK

Prefetching is a well-studied problem in many areas of computer systems. We focus only on recent work on prefetching and code analysis of mobile apps. In [8], the authors propose an API that allows the developer to intelligently prefetch app contents. Our system does not require developers to use new APIs. Moreover, when data is plentiful, we ensure that the user gets the best possible performance – something that all app developers naturally strive for.

Several researchers have focused on modifying mobile apps to improve performance and reduce battery consumption [4, 5, 7]. Reduction in battery consumption may be a side benefit of Procrastinator, but our primary goal is to reduce data consumption.

Researchers have also looked at automatic static and dynamic analysis of call graphs and data flows in mobile apps for various purposes. A well known system is TaintDroid [6], which dynamically monitors the data flow of apps to find privacy leaks. A recently proposed system called ADEL [9] finds energy leaks in mobile apps by dynamically monitoring the data flow in the app. ADEL identifies network calls that are not useful and reports them as energy leaks. In comparison, Procrastinator not only identifies these calls but also automatically delays them to reduce data consumption.

7. SUMMARY

As more cellular operators transition from unlimited data plans to tiered plans, more users will find themselves needing to conserve cellular data consumption. However, many smartphone apps are designed to prefetch network content, in an effort to improve user-perceived latency. Thus, we designed Procrastinator to automatically decide for each network transfer whether it should continue to be prefetched or delayed, based on current need and network cost. This significantly reduces the burden on the app developer, and can significantly reduce the cost for the user. On professionally written apps, our current prototype can achieve as much as 4X savings in bytes transferred, with minimal cost to performance when the user visits non-prefetched content.

8. REFERENCES

[1] Data Sense. http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207005(v=vs.105).aspx.

[2] Fiddler. http://fiddler2.com/.

[3] Procrastinator demo video. http://research.microsoft.com/apps/video/default.aspx?id=202479.

[4] B. Chun, S. Ihm, P. Maniatas, and M. Naik. CloneCloud: Boosting Mobile Device Applications Through Cloud Clone Execution. In *Eurosys*, 2011.

[5] E. Cuervo and et. al. MAUI: making smartphones last longer with code offload. In *MobiSys*, 2010.

[6] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[7] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *OSDI*, 2012.

[8] B. D. Higgins, J. Flinn, T. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Mobisys*, 2012.

[9] L. Zhang, M. Gordon, R. Dick, Z. M. Mao, P. Dinda, and L. Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *CODES+ISSS*, 2012.