Procrastinator: Pacing Mobile Apps' Usage of the Network

Lenin Ravindranath
lenin@csail.mit.edu
M.I.T.
Cambridge, MA, USA

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

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

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

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 app 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 reduction in total bytes transferred by an app, depending on the user and the app. These savings for the data-poor user come with a 300ms median latency penalty on LTE.

1. INTRODUCTION

"This app is too slow" is a phrase that app developers dread when reading user reviews for their apps. Over the last 5 years, smartphone users have forced app developers to optimize their apps for performance. App developers now rely heavily on threading and asynchronous programming patterns to keep their app's UI responsive. They rely on prefetching network content at the launch of an app to hide the network latency of cellular communication.

Recently, low-end system-on-chips from companies including MediaTek and Qualcomm have resulted in significantly cheaper smartphones that are flooding phone markets in the Eastern hemisphere. In the US, we can now buy smartphones outright (without a carrier contract) for \$59-\$99 with a dual-core 1 GHz processor and 8GB storage [3]. This price point, without a requirement to be locked into a large cellular data plan, will attract many users who cannot or do not want to spend money on large cellular data plans. The prefetching nature of apps will hurt these users.

Apps that prefetch network content are downloading 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

MobiSys'14, June 16–19, 2014, Bretton Woods, New Hampshire, USA.
Copyright 2014 ACM 978-1-4503-2793-0/14/06 ...\$15.00.
http://dx.doi.org/10.1145/2594368.2594387.

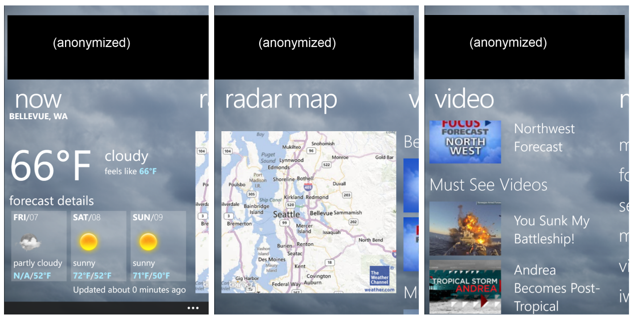


Figure 1: Screenshots of a highly rated and popular weather app. The left screen is visible immediately after app launch. The middle screen is visible after a swipe. The right screen is visible after another swipe.

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 deeper in the app.

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 three sets of users – (1) users that are always conscious about data consumption because they are on a pay-per-byte 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; (3) users that have large data plans at home but are temporarily roaming internationally and are paying additional per-byte charges.

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 Wi-Fi connectivity is free or significantly cheaper) or starts running out of her monthly cellular allotment or is roaming. Alternatively, producing a single adaptive app is difficult for the app developer who has 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.

Our goal is to *automatically* delay prefetching of network content *when appropriate*, to reduce data usage. The system we present in this paper is (aptly) named *Procrastinator*. "Delaying" or "procrastination" of network content is done based on current network connectivity (cellular or Wi-Fi), the status of the user's cellular data plan, where in the app the user is (which parts of the UI are visible), and potential for impact on the 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 want her to write additional code, nor add code annotations. She may need to simply run our system, and can optionally choose to test her app under Procrastinator. 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.

We have implemented Procrastinator for the Windows Phone 8 platform. Our system automatically rewrites app binaries without requiring app source code. 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 rewrite 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, if at all.

We make a number of technical contributions in this paper. Our system automatically identifies for each network transfer which part of the app UI it affects. For those parts of the UI that are not visible, it delays network transfers until they become visible. Our system works on arbitrary third party apps without source code nor app developer effort. At run time, it uses dynamic information about network connectivity and data plan information to automatically switch from prefetching behavior to procrastination. Our lab experiments identify the potential network usage savings in 6 popular, professionally-written Windows Phone apps, as well as additional savings that are untapped because of our conservative approach in avoiding any change to the functionality of the app. Our small study of 9 users with those 6 apps demonstrates savings in total network usage by each app ranging from none to 4X. A larger scale evaluation of 140 apps using automated UI behavior demonstrates 2.5X and higher savings in 20-40 apps, or 20% and higher savings in 60-75 apps. These savings come at no noticeable cost to energy consumption, and additional median latency that is under 300ms.

2. PROBLEM

To understand how app developers prefetch network content, we analyzed the binary code of a large number of apps in the Windows Phone app store. While diverse, the programming patterns used tend to fall into three categories. An example of each is shown in Figure 2. Even though we analyzed .NET byte code, for convenience, we show pseudo-code. Before describing these examples, we note 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 specific UI elements such as a text box.

In Pattern 1, the app developer assigns an image to an image element 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.

Programmers often explicitly handle these asynchronous fetches, especially if additional processing is required before displaying the

```
Pattern 1
void page_load() {
  /* image not visible on the current screen */
  imageElement.Source = http.fetchImage(url1);
}

Pattern 2
void page_load() {
  http.FetchData(url2, callback);
}
void callback(result) {
  var cleanText = clean(result);
  /* text not visible on the current screen */
  textElement.Content = cleanText;
}

Pattern 3
void mainpage_load() {
  http.FetchData(url3, callback);
}
void callback(result) {
  hurricaneWarnings = parseXML(result);
}
/* called when user clicks to navigate to a new page */
void navigate_hurricaneWarningsPage() {
  hurricaneUIElement.Content = hurricaneWarnings;
}
```

Figure 2: Common prefetching patterns in apps.

fetched data. This is illustrated in Pattern 2. The fetched text is "cleaned up"and then displayed by assigning it to a textbox element. The textbox element 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 element 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.

These programming patterns are common because app developers typically lay out and pre-populate most or all of the UI of the app at launch. Even though the user may be viewing a specific portion of the UI (for example, what is visible on a 1024x768 resolution screen), there may be additional content virtually off screen below or to the right or left of the physical screen coordinates, or on different pages. When the user scrolls down a list or slides horizontally, the OS's app runtime will change the viewable coordinates relative to the virtual coordinates of the entire UI. This provides a smooth experience to the user. Unfortunately, this further promotes network prefetching behavior.

It is hard for the app developer to restructure their program to dynamically choose between prefetching and procrastinating. One option is for the app developer to detect at launch whether the user is low on cellular data bytes, and then either layout the entire UI or layout only what is visible. In the latter case, each time the user attempts to scroll up or click on something, she needs to trap that call and then re-layout the screen and present that to the user. This has the disadvantage of significantly increasing code complexity, and having to update two different code paths each time the app developer decides to make even a small change to her UI design.

Alternatively, the app developer could lay out the entire UI, but not tie individual network objects to the individual UI elements. At app launch time, the app would check for network cost. If the network is cheap, it can tie all network objects to all UI elements and prefetch. If the network is expensive, the app developer will have to decide which objects are necessary and only tie those. Each

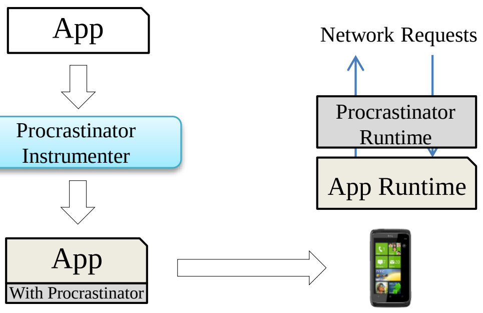


Figure 3: Procrastinator design overview.

```
Setting image source to URL:
<Image Source="http://contoso.com/logo.jpg">

Setting image source to variable that has URL:
<Image Source="{ string variable name }">

Rewriting XAML to detour image setters:
<Image Procrastinator.Source="{ string variable name }">

class Procrastinator {
  DependencyProperty.RegisterAttached("Source", OnSourceChanged);
  void OnSourceChanged(imgElement, url) {
    CheckProcrastinate(imgElement, url);
  }
}
```

Figure 4: Setting image elements using XAML.

time the user makes any move on the UI, such as scroll up a little bit or slide horizontally, the app code will have to re-evaluate which network objects it should fetch, if it has not fetched already. If the network cost changes (such as the user switches to/from Wi-Fi or roaming), then the app code will have to re-evaluate all these decisions again. This adds a significant amount of complexity to the app code.

A third option would be to build a new networking API into the OS. The app developer could specify at the beginning which network objects may be needed, and at different points in the code provide hints to the OS as to whether those objects are definitely needed or no longer needed. We argue that this also adds burden on the app developer – she has to learn a new API and think about every network call and split it into two distinct points in the app code. In the 140 apps that we later evaluate, the number of objects they fetch is 23 on average, and 180 in the highest case. The amount of work that app developers would have to do is not trivial.

We strive for a system that can automatically procrastinate network calls with no app developer effort. We also avoid using app source code. This gives the system the flexibility of being run either by the app marketplace prior to publishing, or by the user, or by the app developer. The trade-offs to explore include data savings versus user responsiveness, delaying network transfers versus energy consumption of cellular radios, and the ability of the app developer to finely control how Procrastinator affects her app.

3. SYSTEM DESIGN & IMPLEMENTATION

We have designed and implemented Procrastinator for Windows Phone apps. As shown in Figure 3, our system consumes an app binary, and outputs a new app binary. It has two main components: the Procrastinator Instrumenter and the Procrastinator Runtime. The Instrumenter takes a Windows Phone app binary and produces a *procrastinated* version. It statically analyzes the app to look for prefetching patterns and finds candidates for procrastination. It then rewrites the associated app code and makes those network calls go through the Procrastinator Runtime.

When producing the new app binary, the Instrumenter will link the Procrastinator Runtime library to the app. When the procrastinated app is run, network calls pass through the Procrastinator Runtime which 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 if the user navigates to the relevant portions of the UI.

3.1 Procrastinator Instrumenter

The mechanism that we use to instrument apps is similar to what is used in AppInsight [19]. AppInsight was designed to find the critical path in user transactions, to help in performance debugging in real-world usage of mobile apps. Procrastinator is instead designed to alter application logic to optimize network usage. Our Instrumenter exploits the fact that apps are often compiled to an intermediate language. While Android apps are compiled to Java byte code, Windows Phone apps are compiled to MSIL [12]. Our Instrumenter works on apps written using the Microsoft Silverlight framework [13], compiled to MSIL byte code. The MSIL byte code representation of an app preserves the structure of the program, including types and methods.

A key challenge in Procrastinator is to automatically find the relationship between network calls and the UI elements that are updated using the data fetched by the network calls. The Instrumenter statically analyzes the app to find such relationships. When such a relationship is found, the Instrumenter rewrites the network call to explicitly identify the UI elements associated with the network call and makes it go to through the Procrastinator Runtime. This takes care of programming patterns 1 and 2 as described in §2. Pattern 3 uses intermediate global variables to store the fetched data before updating the UI elements. In this case, the Instrumenter automatically identifies the relationship between network calls and global variables being set. When this relationship is found, the Instrumenter rewrites the network call to explicitly identify the global variables associated with the network call and makes it go to through the Procrastinator Runtime. We now describe in detail how the Instrumenter rewrites each of these network patterns.

3.1.1 Pattern 1

In programming pattern 1, the app developer is directly connecting the result of a HTTP fetch to an image element in the UI, even though the network call itself is still asynchronous. In practice, this can manifest itself in a variety of ways. The app developer may provide the URL as a hard-coded string, or may use a variable that contains the string. In Windows Phone apps written using the Silverlight framework, this can alternatively be specified in XAML code. An app developer can construct the entire UI of an app in C# code, or using XAML which is a declarative markup language that looks like XML. XAML allows the app developer to separate the UI definition from the run-time logic by using code-behind files, joined to the markup through partial class definitions. This enables the development environment to provide a GUI editor for the app developer to use in designing the UI of an app. Individual UI elements can refer to corresponding variables in C# code. The top of Figure 4 shows the XAML code where the source of an image

```
Pattern 1:
void page_load() {
  /* image not visible on the current screen */
  CheckProcrastinate(url1, imageElement);
}

Pattern 2:
void page_load() {
  CheckProcrastinate(url2, callback, textElement);
}
void callback(result) {
  var cleanText = clean(result);
  /* text not visible on the current screen */
  textElement.Content = cleanText;
}

Pattern 3:
void mainpage_load() {
  /* 2 is a unique id that identifies the network request */
  Procrastinate(url3, 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);
  /* wait does not return until network fetch completes */
  wait();
  hurricaneUIElement.Content = hurricaneWarnings;
}
```

Figure 5: Rewriting code to add procrastination.

element is set to a URL. Another common XAML pattern is to bind the source of the UI element to a variable in the C# program – this is the middle example in Figure 4.

Static analysis: To find pattern 1 through static analysis, the Instrumenter looks for any HTTP fetch API calls that are set to a UI element's source. The value passed into the call can be either a static URL string, or a variable that contains the URL. There are several different image element classes in the Windows Phone SDK, as well as several third party image handlers. The Procrastinator Instrumenter handles all of the common image elements that we have encountered. The Instrumenter also statically analyzes XAML code in the app binary package to find both types of image source setting, and rewrites the XAML so that they can be detoured.

Rewriting: The Instrumenter replaces the network call in pattern 1 in Figure 2 with the `CheckProcrastinate` call as shown in Figure 5. The rewritten call explicitly passes the UI element with the url. In the case of XAML, we rewrite the XAML to provide a callback whenever the url is bound to a UI element. We achieve this using the Dependency Property feature of the Windows Presentation Foundation, which can be used to extend the functionality of a common language runtime (CLR) property. Essentially, it gives our code a callback into a user defined function when a property of an element is changed. During the callback, the parameter to the function is the UI element being set and the property value that is bound (URL in this case). We alter the XAML code to trap the image setting and call `CheckProcrastinate` with the two parameters, as shown in the bottom of Figure 4.

3.1.2 Pattern 2

In pattern 2, the developer passes a callback function to the network call. The network call asynchronously fetches data and invokes the callback with the data. The UI elements are then accessed and updated in the callback, as shown in Figure 2. Instrumenting this pattern is more challenging. First, statically identifying the UI elements that are updated is not trivial. In the example shown, the callback method itself updates the UI element. However, the callback method could call other methods (perhaps asynchronously!) and the updated UI elements may lie deep in this call tree. 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 write to the file system that another thread reads. If there are any side effects, then those network calls are not candidates for Procrastinator, and we allow them to proceed as normal.

category	num
no side effect	8,869
changes UI	1,053
changes input parameters	275
changes property of instance that the method is called on	2,228
has side effect	756

Table 1: The five categories that we classified each Windows Phone SDK API, system call and popular third party library calls into, and how many are in each category.

Static analysis: This involves the following steps:

1. find the callback method associated with the call
2. generate a *conservative* call graph (one that includes all possible code paths) that is rooted in the callback method. This call graph contains both synchronous and asynchronous calls.
3. analyze code in the conservative call graph to discover all UI elements being updated
4. analyze code in the call graph to ensure it has no side effects. If any code in the graph has any side effects, the Instrumenter conservatively decides that the call cannot be procrastinated, and the code is not modified.

Call graph: Generating a call graph in the presence of virtual method calls is challenging. At compile time, the type of the instance for which invocation takes place is not known. At run time, when invoking a virtual method, the run-time type of the instance determines the actual method to invoke. To overcome this problem in static analysis, we take a conservative approach. In the call graph, we include *all* the potential calls in the class hierarchy if we are not able to statically resolve the object type reference. Similarly, when a function pointer is used, we include all the potential calls that the function pointer could point to, since we may not decisively know what the contents of that function pointer are during static analysis. When an asynchronous call is made, we identify the potential callback and recursively include the call graph of the callback.

Side effects: Our analysis of all the Windows Phone marketplace apps shows that there are over 13,000 unique system calls, SDK API calls, and third party library calls, that apps use in the call graphs of network callbacks. These range from methods such as Math.Sqrt to VibrateController.Start. Some methods such as Math.Sqrt do not modify any input parameters nor modify the state of the phone, such as the filesystem. Those calls are free of "side effects". Other calls may modify input parameters, which may be global variables that other threads in the app may interact with. We manually sorted these calls into the five categories in Table 1. We conservatively assume that any call that changes an input parameter or class instance that is globally accessible by other threads is not side effect free. If a call graph for a network callback has any system call or SDK call that has a side effect, the Instrumenter ignores that network call for procrastination and leaves it untouched. Each time the API list available to apps changes, which typically happens when there is a major OS update, this categorization will need to be updated.

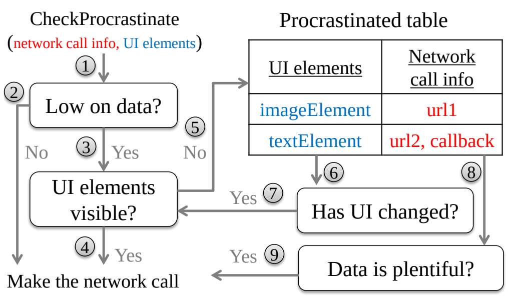


Figure 6: Procrastinator Runtime tracking UI elements.

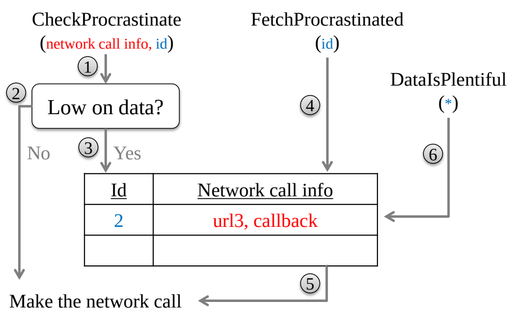


Figure 7: Procrastinator Runtime tracking global variables.

Rewriting: For network callbacks that do not have a side effect, the static analysis outputs a list of UI elements that are updated by a network call. The network call is rewritten as shown in Figure 5. We explicitly pass the UI elements with the network call information into the Procrastinator Runtime, similar to pattern 1.

3.1.3 *Pattern 3*

In pattern 3, the result of the network request is stored in memory (global variables) and used later. This may be done to prefetch data for UI elements that have not yet been instantiated. Here, the Instrumenter's goal is to find the relationship between network calls and global variables.

Static Analysis: We do two passes of static analysis for this pattern. In the first pass we analyze network callbacks to identify global variables used in *store* operations. In the second pass, we analyze the rest of the app code to identify code where the same global variables are accessed in *load* operations. The first pass analysis is similar to pattern 2. We generate a conservative call graph of the callback and analyze the code in the entire call graph. Instead of looking for UI elements, we look for *store* operations to global variables. Note that we do not consider these store operations to a global variable as a side effect here. We do look for other side effects such as accessing system APIs that affects the phone state, such as the filesystem. If there is a side effect other than updating global variables, we do not consider it for procrastination. In the second pass, we analyze the entire app to find load operations of the same global variables that was identified in the first pass. We instrument both the network call and these load points as follows.

Rewriting: We rewrite the network call to go through the Procrastinator Runtime as shown in Figure 5. Along with the network call information (url and callback), we pass in a unique identifier for the set of global variables stored by the callback. Just before where these global variable are loaded, we instrument the code to inform the Procrastinator Runtime to fetch the network data before it is accessed. The network call to execute is identified by the unique identifier.

3.2 Procrastinator Runtime

The Procrastinator Runtime behavior for patterns 1 and 2 is summarized in Figure 6 and pattern 3 in Figure 7.

Patterns 1 and 2: `CheckProcrastinate` calls into the Procrastinator Runtime with the network call info (URL and UI elements, and network callback method for pattern 2). The Runtime checks if the user is low on data (1), described below in §3.2.1. If not, the network call is not procrastinated and issued right away (2). Otherwise, the runtime checks the visibility of the UI elements being passed (3), described below in §3.2.2. If at least one UI element is visible, the network call is not procrastinated and continues to execute (4). If the relevant UI elements are not visible on the current screen, then the network call is procrastinated (5).

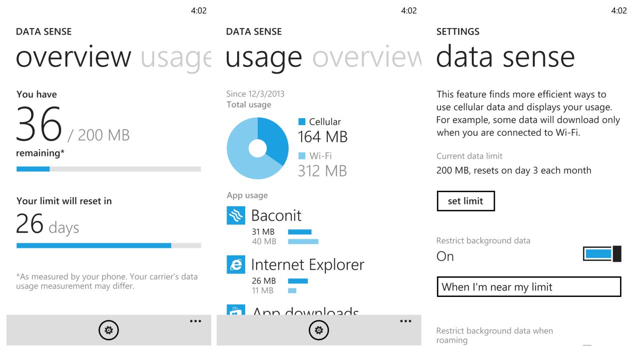
Procrastinated network call info and associated UI elements are added to the procrastinated table. This in-memory table maintains requested network calls that have not been issued. Whenever the UI changes (6), described below in §3.2.3, the runtime iterates over this table and checks the visibility of each UI element in it. If any UI element becomes visible, the associated network call is made and the entry is removed from the table (7). The runtime also tracks changes in network connectivity. If data becomes plentiful, *all* outstanding network requests are immediately issued (8).

Pattern 3: `CheckProcrastinate` calls into the procrastinator runtime with the network call and the unique identifier (1). If the user is not low on data, the network call is made right away (2). Otherwise, the call is procrastinated and added to the procrastinated table indexed by the unique id (3). The call is delayed until one of the global variables is accessed by any thread in the app (4). We intercept all accesses to the identified global variables and pass the same unique identifier to inform the Procrastinator Runtime to fetch the request before the global variable is first accessed. Since all network calls in the Windows Phone SDK are asynchronous, we have to instrument the thread to wait until the transaction is complete before proceeding. When `FetchProcrastinated` is called, the network call is removed from the table and executed. Again, if data becomes plentiful, *all* outstanding network requests are immediately issued (6).

3.2.1 *Checking network connectivity and cost*

Procrastinator makes use of a feature called Data Sense that we previously helped build and deploy inside the Windows Phone 8 OS. As shown in Figure 8, it allows the user to enter their cellular data plan information: type (such as monthly or pay-per-byte), limit (such as 200 MB), and reset date (such as 3rd of every month). The OS carefully tracks every byte sent and received over the network and attributes it to the responsible app or OS feature. The user can see the per-app breakdown in the UI, and request that the OS automatically restrict data consumption when nearing the data cap. The OS tracks overall consumption against the data limit and advertises changes in available data to OS services and apps [2]. The app can query (or be alerted) when the network cost type changes ("unrestricted" when on Wi-Fi or unlimited cellular data, "fixed" when on periodic plans, or "variable" when on pay-per-byte plans) and when available data changes ("ApproachingDataLimit", "OverDataLimit", or "Roaming").

Each time a procrastinated app requests a network transfer, the Procrastinator Runtime checks the current network cost type and properties to determine whether procrastination is warranted. If the network cost type is "unrestricted", or "fixed" and not any of "Ap-



You have 36 / 200 MB remaining*

Your limit will reset in 26 days

*As measured by your phone. Your carrier's data usage measurement may differ.

Since 12/3/2013
Total usage
Cellular 164 MB
Wi-Fi 312 MB

App usage
Baconit 31 MB 40 MB
Internet Explorer 26 MB 11 MB
App downloads

This feature finds more efficient ways to use cellular data and displays your usage. For example, some data will download only when you are connected to Wi-Fi.

Current data limit
200 MB, resets on day 3 each month

set limit

Restrict background data
On
When I'm near my limit

Restrict background data when roaming

Figure 8: Screenshots of Data Sense on Windows Phone 8.

proachingDataLimit", "OverDataLimit", and "Roaming", then the transfer will be allowed. Otherwise, only transfers that affect currently visible UI will go through. Any time network cost changes, the Procrastinator Runtime gets an event up-call from the OS, and will reevaluate all procrastinated network calls. On OSes without Data Sense, we imagine a UI button can be exposed to the user to manually turn procrastination behavior on or off for all apps.

3.2.2 *Checking visibility of UI elements*

To check if a UI element is visible to the user, the Procrastinator Runtime extracts the current UI tree and traverses it to find the location of this UI element in the tree. We use the size of the UI element, position in the UI tree, the size of the UI page, the resolution of the screen, and current view coordinates to check if it is visible to the user. In most cases, the size or bounding box of the UI element is defined in the app code (such as an image of 500x300 pixels). In these cases, straightforward math can determine if any portion of the UI element is visible to the user. In relatively rare cases, the size of the UI element is not defined by the app code and is instead determined by the downloaded network object. In these cases, if the center of the UI element is not visible, we assume that the UI element is not visible.

3.2.3 *Checking if the UI has changed*

All procrastinated requests are reevaluated every time the UI changes. To detect UI changes, the Procrastinator Runtime includes event handlers for UI manipulation events and UI update events. Manipulation events are fired whenever the user interacts with the phone (such as click or swipe). Update events are fired whenever a UI element is updated programmatically. When one of the events fires, the runtime iterates over each procrastinated request and checks if any of the UI elements associated with the request is visible. When the procrastinated table is empty, the Runtime will de-register from these event notifications, and re-register if any are later added.

3.3 Implementation

The Procrastinator Runtime is written in 839 lines of C# code, which compiles down to a 46 KB library of MSIL code. This library is included in all procrastinated apps. The Procrastinator Instrumenter, which consumes an app binary and produces a procrastinated app binary, is written in 861 lines of C# code, not including the Microsoft Research Common Compiler Infrastructure libraries [6] that it relies on to parse MSIL code and replace calls.

In practice, many apps and programming constructs rely on virtual method invocations and function pointers at the MSIL level. As a result, when dealing with programming patterns 2 and 3, our Instrumenter tends to construct large call graphs for the network

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 2: The six Windows Phone apps we evaluate and the action we perform in each run of the app in lab experiments. Each app is one of the top apps in its category, and is authored by the professional online company that owns both the app and the primary web service that it relies on (e.g. Twitter by Twitter, Inc.).

callbacks. The call graph is large because we conservatively include any possible method that might match the virtual method call or function pointer. The problem with a large call graph is that the chances of encountering a system call or SDK call that has a side effect go up significantly. As a result, in practice, the Procrastinator Instrumenter will procrastinate almost all image transfers because they tend to follow programming pattern 1. Network calls that download non-image content tend to follow programming patterns 2 and 3, and those rarely get procrastinated due to the risk of unintentionally altering app behavior. We expect that images account for the bulk of network bytes and will provide significant savings in our evaluation.

4. EVALUATION METHODOLOGY

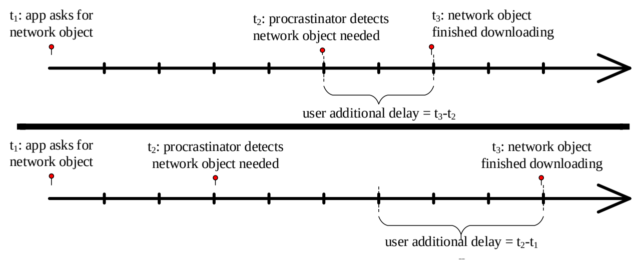
To evaluate Procrastinator, we use a three-pronged strategy. We begin by evaluating 6 popular apps in the lab. By manually inspecting every network transfer and the MSIL code for these 6 apps, we can understand the behavior of these apps and of their procrastinated versions. Next, we deploy these 6 apps to 9 users in a small user trial to understand savings that Procrastinator offers in more realistic usage. Finally, we evaluate Procrastinator using a much larger set of 140 apps in Windows Phone emulators using UI automation.

4.1 Lab experiments

To test the effectiveness of Procrastinator, we pick 6 apps from the top 50 in the Windows Phone marketplace, listed anonymously in Table 2. We installed these 6 original apps, as well as their procrastinated versions on a Nokia Lumia 920 smartphone. 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. We configured the phone so that all HTTP and HTTPS traffic was intercepted by a server with the Fiddler [1] proxy.

We ran both versions of each app, performed the action listed in Table 2, and captured traces of network activity. 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 – we turned off this feature here.

We manually inspected the Fiddler traces and compared the content that was fetched to what was displayed on the screen. Apart from verifying what appeared on the screen, we also carefully looked through the app's binary code to determine whether the object was needed to render the page that the user saw. 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 2. In cases where we cannot deduce whether



t_1: app asks for network object
t_2: procrastinator detects network object needed
t_3: network object finished downloading
user additional delay = t_3-t_2
t_1: app asks for network object
t_2: procrastinator detects network object needed
t_3: network object finished downloading
user additional delay = t_2-t_1

Figure 9: Two examples (top and bottom) of how we calculate the additional delay experienced by a user in a procrastinated app.

the content of web objects is used in the screen, typically when we cannot decipher the web object and the relevant MSIL code, we conservatively assume them to be "necessary". If a web object is not necessary, then it is either "skipped" or "prefetched". If Procrastinator correctly deduces that a network object is not necessary and does not download it, we count this as "skipped". If Procrastinator allows an object to be downloaded that is not necessary, we count this as "prefetched". This happens when Procrastinator cannot determine that not fetching a network object will not cause an unintended side effect in the behavior of the app (such as playing music or vibrating the phone or writing to the filesystem), and hence conservatively allows the network object to be prefetched.

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. These savings can come at the cost of additional delay experienced by the user when a procrastinated network object has to be fetched, such as in App6 in Figure 1, if the user swipes to see the radar image. This additional delay depends on a variety of factors, including the speed of that network, the size of the network object, and how responsive the web server is.

To evaluate these aspects, we deployed the procrastinated versions of these 6 apps to 9 colleagues for 2 months [1]. We asked the users to use any of these 6 apps that they like as normally as they would if they discovered them in the app marketplace. Over this 2 month period, we collected a total of 553 sessions across these users and apps where their phone had network connectivity.

All 9 users have unlimited data on their LTE cellular plans. Since the phones are never in danger of exceeding their data limits, for the purposes of our study, we turned off the Data Sense network check so that Procrastinator would always attempt to procrastinate. We also write a log for every user session. This log records when the user launches the app, what network transfers the app requests, which ones Procrastinator allows through, which ones it procrastinates, when (if at all) a previously procrastinated request becomes necessary to display to the user, and the sizes and end times of all network transfers. To determine the size of any requests that are procrastinated and never fetched by Procrastinator, we download those objects on the side simply to check their size for our evaluation graphs. These logs are periodically uploaded in the background to the Microsoft Azure cloud.

[1]This user study was performed in accordance with Microsoft Research privacy guidelines, record #1903.

parameter	value
LTE promotion power	1210.7 mW
LTE promotion duration	260.1 ms
LTE continuous downlink transfer power	1950.1 mW
LTE tail power	1060.0 mW
LTE tail duration	11576.0 ms

Table 3: Parameters of LTE energy model from Tables 3 and 4 of the 4GTest paper in ACM MobiSys 2012. We assume downloads happen in LTE continuous reception mode and we use the power values for median throughput of 12.74 Mbps.

Using these logs, we calculate how much network activity occurred in each user session of each app. We calculate how many bytes were fetched that the app requested that Procrastinator did not interfere with, and label these as "fetched" bytes. We calculate how many bytes Procrastinator did not fetch, and the app did not display to the user – these are "skipped" bytes, and represent our savings. Finally, there are bytes that Procrastinator did not initially fetch, but later detects that they are needed and subsequently fetches, thereby incurring additional delay – these are "delayed" bytes.

For "delayed" bytes, we want to evaluate how much longer the user had to wait as a result of our decision to not initially fetch the network object when the app requested it. Figure 9 shows two example situations that can occur. At time t_1, the app requests a network object that Procrastinator decides not to fetch because the relevant UI is not visible to the user. At a later time t_2, Procrastinator detects that the relevant UI is shown to the user and starts fetching it. This transfer completes at time t_3 and the UI is updated. In the top example, the download time for the object $t_3 - t_2$ is smaller than $t_2 - t_1$ which is the time between the initial request by the app and when Procrastinator starts the fetch. In this case, the *additional* delay that the user experiences is the download time of $t_3 - t_2$. In the bottom example, the download time of $t_3 - t_2$ is longer than $t_2 - t_1$. In this case, the *additional* delay that the user experiences is $t_2 - t_1$, which is the extra delay introduced by Procrastinator.

Finally, we want to evaluate how Procrastinator impacts the phone battery consumption of apps. When Procrastinator does not fetch unnecessary bytes from the network, it should reduce energy consumption by cellular radios since that depends on how much the radio is transferring and how long it is awake for. However, when Procrastinator delays the transfer of network objects, it can potentially increase energy consumption – the radio may have gone to sleep due to network inactivity and may need to spend energy waking up, and now the radio may be up for longer than when network objects were batched under prefetching behavior. Since it is difficult to accurately pinpoint the energy consumption of an LTE radio at fine time granularity and what energy state it is in while an app is running in the hands of a user, we use the empirical energy model constructed by prior work [10]. Table 3 lists the energy model parameters that we use. For each user session of an app, we construct two time series. One time series identifies when the app would normally fetch objects from the network, and the other time series identifies when the app would fetch objects with Procrastinator. By factoring in transfer times, and LTE radio promotion time and tail duration, we calculate the total energy expenditure of the LTE radio in each user app session, assuming the radio is off when the app is launched. We include the LTE tail power consumption both during the session, if the radio would go to sleep, and at the end of the session.

monkey	functionality
M1	for up to 15 minutes, random swipes and clicks
M2	for up to 1 minute, random swipes and clicks
M3	launch app, wait for all transactions to finish, quit

Table 4: The 3 automated user profiles (or "monkeys") that we use to evaluate 140 apps in the Windows Phone emulator.

4.3 Automated monkeys

Finally, to understand the data savings and latency penalty of Procrastinator on a broader set of apps, we use UI automation (or "monkeys"). From all the apps in the Windows Phone marketplace (across the entire 5-star rating spectrum) that make at least one network call, we pick a random set of 140 apps. We then apply UI automation to these apps to emulate user behavior. We run these apps in the Windows Phone emulator running on a server. We disable the Data Sense check by Procrastinator, so that it always assumes the phone is on cellular and the user is running out of data plan bytes. We continue to log actions by Procrastinator, and any network objects that are skipped are fetched on the side so that we can calculate their sizes for our graphs.

Table 4 lists the three monkeys that we use. M1 will launch an app and aggressively browse it in a random pattern. It will click on icons and dialog boxes and other UI elements and it will scroll horizontally left and right, but it does not scroll up or down. It will do this for up to 15 minutes, unless the app quits or crashes prematurely. We consider this to represent users that will explore most of an app's UI, but not necessarily the entire UI. M2 is similar to M1, except that it does not run for more than 1 minute. M3 is the least aggressive – it will launch an app, wait for any pending computations to finish, and then exit the app. M3 will explore the least amount of the app's UI, and represents user behavior where the main page of the app displays the information relevant to the user, such as in weather apps. In comparing monkey runs to our user trials, M2 closely matches user behavior with respect to data consumption, and saved and delayed bytes. Details of how our UI automation system works are in [18].

We run each monkey on each app 15 times, though we may get fewer useful logs if the app crashes occasionally on launch. Unlike in our user study where we focused solely on top, professionally-written apps, here we include apps from across the spectrum of user ratings, some of which are buggy. Using logs from these runs, we calculate how many bytes were fetched, skipped, and delayed. The Windows Phone emulator that we run the monkeys on is connected to Ethernet. Hence latency numbers for delayed objects do not make sense, unlike in our user study where users were sometimes on LTE. Instead, we calculate what the latency would be to download delayed objects if the network connection was LTE. We use the median values for LTE throughput and latency from Figure 5 of prior work [10], specifically 12.74 Mbps downlink on LTE and 69.5 ms RTT. We calculate latency while factoring in whether the radio was still connected and awake as a result of prior transfers or asleep, TCP connection setup delay, HTTP request and responses, TCP window ramp up, and whether an existing HTTP connection was open as a result of a prior fetch from the same server. Unlike in the user study, *additional* latency does not make as much sense in this context, so we present the full latency to download delayed objects.

In addition to the LTE model, we also consider a "cloudlet" model. Cloudlets [20] provide an interesting opportunity here. If there is a cloudlet running in the celltower, it can fetch objects that Procrastinator skips on the phone. In case the user does go to that part of the UI, the network object is now sitting at the nearby cloudlet, instead of on a distant webserver. In this way, the latency of fetching delayed objects is significantly reduced, while still preserving spectrum usage over the air for objects that are not needed at all. In this model, we assume that the phone has a low latency connection to the cloudlet that is still limited on throughput – 12.74 Mbps downlink and 3 ms RTT, while the cloudlet suffers the remainder of the end-to-end RTT latency of 66.5 ms, but is not limited by air spectrum and has 50 Mbps throughput to the Internet. When Procrastinator decides to skip a network object, it sends that URL to the cloudlet, which fetches it. If Procrastinator detects that the object is needed, it fetches it from the cloudlet. We calculate the download latency for the cloudlet, and the download latency between the phone and the cloudlet. In the best case, the initial download to the cloudlet will have finished before the phone needs it, in which case the penalty is the latter download latency. In the worst case, the phone has to wait for the cloudlet to download it and then download it from the cloudlet. While cloudlets do not exist today and hence this depends on a number of assumptions, we nonetheless want to consider how Procrastinator would fare in this future model.

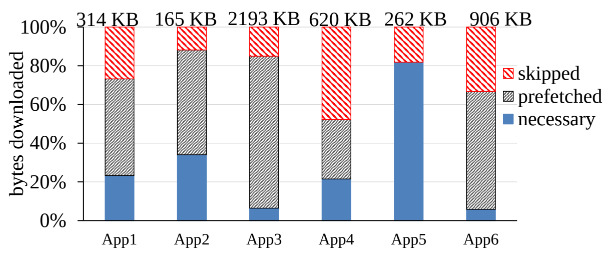


Figure 10: Bytes consumed when running each of the 6 original apps, and their procrastinated versions.

5. EVALUATION RESULTS

We now present results from our evaluation. We begin with our in-lab experiments on 6 popular apps, followed by our small user deployment of the same set of apps with 9 users. Finally, we examine a broader set of 140 apps that we evaluate using UI automation.

5.1 Lab experiments

Our lab results are summarized in Figure 10. For all 6 apps, the unnecessary content dominates the downloaded bytes when performing the actions listed in Table 2.

In Table 5, we list the network objects that the original App6 requests on app launch. It requests 31 web objects, for a total of 906 KB. Refer back to Figure 1. 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 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 is only interested in the current weather and forecast, and hence remains on the first screen and does not visit the other screens, then the radar map and the video images are all wasted downloads. Procrastinator correctly identifies those as wasted because the screen coordinates for those images are off screen, and hence does not download those 5 objects listed at the bottom right of Table 5. This corresponds to 301 KB out of 906 KB that are skipped by Procrastinator, leading to roughly 33% of savings for the user.

The app also requests some large XML files that describe photos that people have taken of weather phenomenon across the US and

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 5: 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.

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. These objects add up to 552 KB and involve programming patterns 2 and 3 from Figure 2 where static analysis of the app's callback methods could not guarantee that there is no side effect from not downloading these network objects. As a result of our conservative design to avoid any potential for change in app behavior (other than network delays), Procrastinator allows the app to download these objects. These are marked as "prefetched" in Figure 10. Ultimately, this app needs only 52 KB of network objects to show the contents of the first screen to the user.

Other apps are similar. App1 downloads 40 objects. Some are text descriptions of recipes and some are photos of recipes. 10 of these objects are necessary, 6 large objects are skipped by Procrastinator, and another 24 small objects bypass Procrastinator and are still prefetched by the app. App3 downloads several web objects, most of which are small XML blobs describing each movie, review details, news and user comments. App4 downloads news articles and news images, of which Procrastinator skips 303 KB, while the app continues to prefetch 195 KB that it does not need, and 136 KB that it does need, leading to 48% of savings. App5 downloads fewer objects, but does prefetch some large ones, all of which Procrastinator skips.

5.2 User trial

The savings we demonstrate in the lab are highly dependent on what the user does in the app. To understand this in a more realistic setting, we now present results from our deployment of the same apps with 9 users. Figure 11 shows the number of app launches across all users for each app. Our users seem to check the weather more frequently than they cook with a new recipe. We collected data for 553 user sessions in total.

Figure 12 shows the number of bytes spent per user session. Note that the total bytes requested by an app varies across runs, such as when the radar map image for Seattle is of a different size than the radar map image for San Francisco, or when content is cached by the app. We see that in the case of App6, Procrastinator typically reduces the number of bytes by over 4X. In the other extreme, for App2, Procrastinator almost never saves any bytes, perhaps because users scroll through all available movies before leaving the app. In this situation, Procrastinator hurts because any objects that it decides to not prefetch, it will later need to fetch when the user visits that part of the UI, resulting in "delayed" bytes. We also note that for some apps, such as App3 (which shows the user more photos of each movie), there is huge variance in bytes saved.

We contrast this with Figure 13, where for App6, a small number of objects are skipped, but those objects are large and contribute a lot to the savings we achieve. App3 has a lot of small images (stills

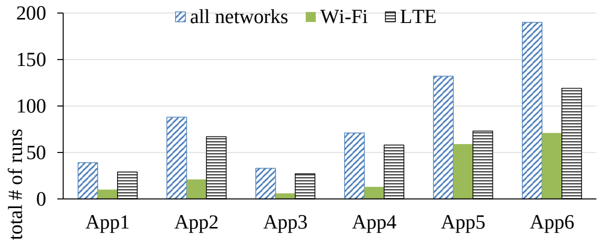


Figure 11: Total number of user sessions per app.

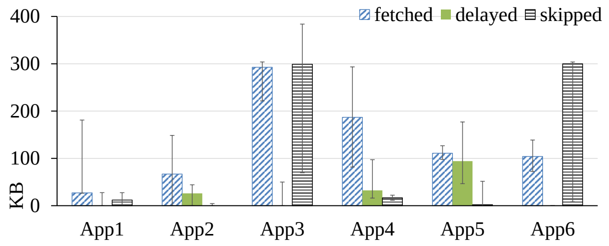


Figure 12: Median KB fetched, delayed and skipped per session, with 25th and 75th percentiles, across all user sessions.

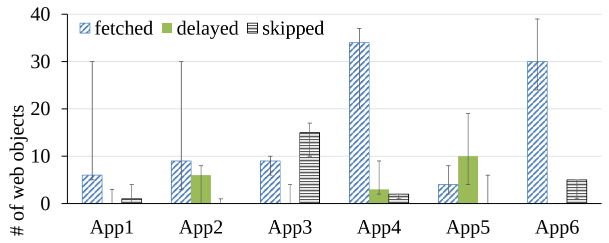


Figure 13: Median number of web objects fetched per session, with 25th and 75th percentiles, across all user sessions.

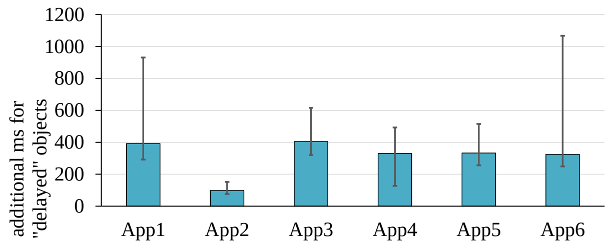


Figure 14: Median additional delay incurred per delayed web object, with 25th and 75th percentiles, across those objects incorrectly procrastinated in user trial, when user was on a cellular connection.

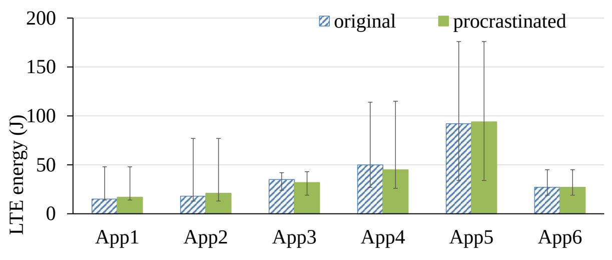


Figure 15: Median energy spent on LTE radio in original and procrastinated behaviors of apps, with 25th and 75th percentiles across all user sessions.

from movies), and procrastinating many of those small objects add 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 cases where the user is shown a skipped image, we calculate the *additional* delay that she incurs as a result of Procrastinator. 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 14 shows this additional delay. We find this is typically under 1 second, and under 300ms in the median across all app sessions. Recall that this is a penalty that a user will experience only when she is on a cellular connection, and running low on her data plan bytes, and visits app content that is not on the first screen of the app. For users in this situation, we believe this is a compelling trade-off.

While we cannot directly measure the energy consumed by the LTE radio on the smartphones that our users used, we apply an empirical model from prior work [10] using the actual timings of network transfers from the user logs. In Figure 15 we show the energy consumed by the app with normal behavior and under Procrastinator. The energy consumption is largely similar. The energy benefit of not transferring unnecessary bytes is balanced by the energy cost of waiting to transfer bytes that later become necessary. We notice that App5 exhibits high variance, partly because users spent variable amounts of time reading articles before clicking on the next article, sometimes as long as 12 seconds which can trigger the LTE radio to sleep.

5.3 Automated monkeys

To understand the performance of Procrastinator in a wider set of apps, we use UI automation to run 140 apps multiple times in the Windows Phone emulator. In Figure 16, we show the number of bytes that were fetched, skipped and delayed when using the most aggressive monkey, M1. In contrast, Figure 17 shows that for the least aggressive M3. Clearly apps consume fewer bytes under M3 (note the different scales on the vertical axes). Even though M3 launches the app and quits soon after without interacting the app's UI, it is possible for there to be delayed network objects. Procrastinator may decide to not download an object, then the app code automatically changes the UI screen despite not having any input from the user, at which point Procrastinator detects the need for previously requested network objects which it will then download.

Figure 18 shows CDFs of the fraction of bytes saved by the three monkeys across the 140 apps. We see that with M3, Procrastinator saves over 60% of bytes in 40 apps. With the most aggressive monkey, it saves over 60% of bytes in 20 apps, and over 40% in 40 apps. There is a tail of over 50 apps where the savings are practically nothing, but these apps consume relatively less network data. The apps that represent points to the left of 90 on the horizontal axis consume 1.1 MB on average, while the ones to the right consume 475 KB on average.

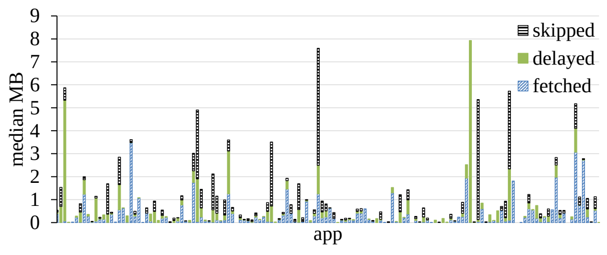


Figure 16: Median MB skipped, delayed and fetched across 140 apps, when using monkey M1.

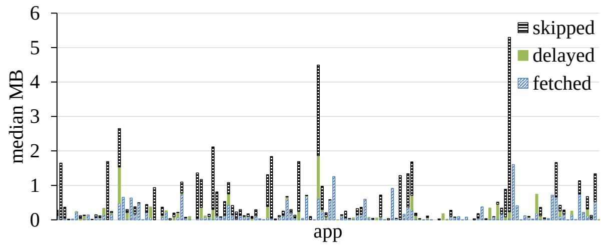


Figure 17: Median MB skipped, delayed and fetched across 140 apps, when using monkey M3.

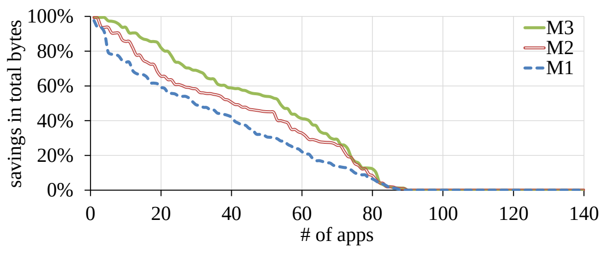


Figure 18: CDF of % of median bytes saved when using Procrastinator on 140 apps. Each line is an independent CDF of a different monkey. Vertical axis is (bytes saved) / (bytes saved + bytes fetched + bytes delayed).

Figure 19 shows the median latency in downloading delayed objects under an LTE model, and Figure 20 shows that assuming a cloudlet at the LTE celltower. Under M3, almost half of apps experience no delay because there is less opportunity for there to be delayed objects in M3. There are some objects that take more than 500 ms to download – these are all large transfers. In some cases, they are 1 MB files, which is almost a second at 12.74 Mbps, which is sometimes exacerbated if the radio went to sleep due to inactivity or there was no open TCP connection that server. In the case of LTE, we observe that all non-zero latencies start at about 100 ms. This is because the LTE end-to-end RTT is 69.5 ms and if an HTTP connection is already open to a server, it can take a little more than a round-trip to get an entire small object. In contrast, Figure 20 does not show this artifact. In this hypothetical model of a cloudlet at the celltower, the radio latency to the cloudlet is much lower,

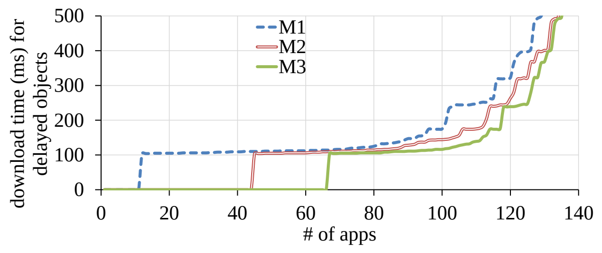


Figure 19: CDF of median download time for delayed objects in 140 apps under Procrastinator when using monkeys. Download times are calculated using LTE parameters from prior work. Each line is an independent CDF representing a different monkey. Vertical axis is chopped at 500ms for readability.

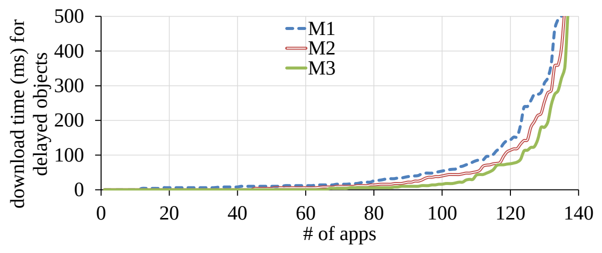


Figure 20: CDF of median download time for delayed objects in 140 apps under Procrastinator when using monkeys. Download times are calculated assuming a Cloudlet at the LTE celltower. Each line is an independent CDF representing a different monkey. Vertical axis is chopped at 500ms for readability.

while the cloudlet continues to experience the bulk of the latency to the Internet. In this way, the cloudlet can hide network delays.

5.4 Overheads

Due to the addition of the Procrastinator Runtime and the code injection in the app's binary, the size of the app package increases. This can impact users in two ways. The bytes spent downloading the app from the marketplace will increase. The bytes consumed by local storage on the phone in storing the app binary will increase. In Figure 21, we present the increase in size of the app package. The compressed line shows the increase in size of the compressed package, which is what the phone will download from the marketplace. The uncompressed line indicates how much storage will be consumed on the phone. The median app size before Procrastinator is 1.6 MB uncompressed and 813 KB compressed. The largest is 29 MB uncompressed and 25 MB compressed. All of the apps with more than 5% increase in uncompressed size were under 1MB in size uncompressed prior to Procrastinator. All of the apps above 10% are under 544 KB in size. While it is unfortunate that using Procrastinator will increase the size of the app that is downloaded by the user and hence consume additional cellular bytes, this minor increase is easily amortized if the user runs the app frequently and thereby saves cellular bytes then.

Another overhead that Procrastinator introduces is the memory consumption in maintaining the two tables that track procrastinated calls as shown in Figures 6 and 7. The number of rows in the tables in any app is bounded by the number of prefetch network calls that the app makes. Each row consists of the URL being procrastinated and any associated callback method reference or UI element reference. Similar to Figure 13, we do not see an excessive number of downloads in any run of an app – it is typically well below a hundred. Hence, the memory consumption of these tables is minor in practice.

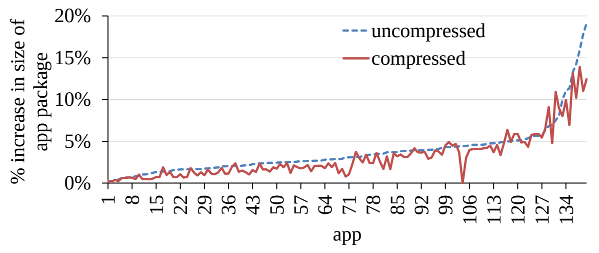


Figure 21: Increase in size of app package as a result of Procrastinator, for 140 apps. Highest point is almost 20%. Apps are sorted on the horizontal by uncompressed increase percentage.

UI tree depth	average delay
3	0.1 ms
13	0.2 ms

Table 6: Delay in checking whether a UI element is visible on the phone's screen. Delay reported is the average of 1000 runs on a Nokia Lumia 920 smartphone. UI depth is the number of levels in the UI tree that the UI element is at.

The Procrastinator Runtime introduces very little delay during program execution. The most common code execution is to check the visibility of UI elements (§ 3.2.2), which may happen each time an app requests a network object, and once per procrastinated network object each time the UI changes. This code traverses the UI tree and does simple math to determine the visibility of each parent node of the UI element in the tree. In Table 6, we evaluate the overhead of this check for two common UI tree depths. In each case, we calculate the time spent in running this code path 1,000 times and then divide that by 1,000. We find that running this code requires 0.1 – 0.2 ms. If an app has 100 UI elements that are procrastinated (far more than we have observed in practice), this overhead will be 10 – 20 ms each time the UI changes.

6. DEPLOYMENT IMPLICATIONS

Here we summarize our thinking on a number of practical issues that one would consider in a public deployment of Procrastinator.

Balancing speed and cost: Our target user for Procrastinator is one who frequently switches between being data cost conscious (when on pay-per-byte data plans or roaming internationally) and data cost oblivious (when on Wi-Fi or at the start of a monthly billing cycle). Procrastinator offers a compelling trade-off of occasional delays in network transfers for data cost savings for these users. There is a middle ground that our system could support. Based on prior usage history, existing machine learning algorithms could learn that a particular user often goes to certain portions of an app's UI. With that knowledge, Procrastinator could choose to fetch additional content, thereby reducing the user experience delay, at the risk of occasionally being wrong and consuming more bytes than needed. In either case, users who prefer to manually turn off Procrastinator can do so using the existing button in Figure 8.

Request prioritization: While our goal is to reduce data consumption, Procrastinator's techniques can be used to improve network performance. Apps tend to intermingle urgent network requests with non-urgent prefetches. Procrastinator automatically identifies which network requests are prefetches. It could hold off on those until the network becomes idle. In this way, prefetch transfers will not compete with more urgent requests for precious available bandwidth.

Impact on server: Procrastinator affects only HTTP(S) GETs. Other traffic, including UDP, ICMP, HTTP POST, and (non-HTTP) TCP remain unaffected by Procrastinator. Semantically, HTTP GET is a mechanism for retrieving content from a server. The same cannot be said of arbitrary TCP or UDP traffic, and certainly not for HTTP POST which is used to commit an action at a server (such as purchase a movie ticket). Hence we do not procrastinate any such traffic. One could design a client-server system where an HTTP GET actually signals an action or commit on the server side. So far, we have not encountered such unusual behavior in the apps we have studied. If that were to happen, and if Procrastinator were to procrastinate such a call, that would affect the broader semantics of the application. It is possible that by not fetching some HTTP objects, Procrastinator may impact analytics done on server-side HTTP logs.

Who runs Procrastinator? We designed Procrastinator to not require mobile OS changes, not require app developer effort, and not require app source code. These design decisions give us significant flexibility in deploying Procrastinator. We can consider three models:

- Developer tool: The app developer can run Procrastinator on their app binary, test the new app binary, and submit that to the app marketplace.
- Marketplace tool: The app marketplace could run Procrastinator on incoming app binaries as part of the app publication process. The marketplace may chose to do this selectively for users, or countries where data costs are a significant concern for the population.
- User tool: The data-conscious user could run Procrastinator on app binaries that she has purchased from the marketplace.

We do not advocate any one model over another. Each group has their own incentives. Some users want to save data costs. An OS vendor may want their phones to be appealing to data conscious users. An app developer may want to support data conscious users without hurting performance demanding users.

Procrastinator versus "Prefetcher": In our experience, most apps prefetch network content, such as images and icons and XML files. Thankfully, apps typically do not prefetch large video files, and will wait for the user to click on a video. Procrastinator works because apps prefetch. Hence our system adaptively procrastinates those prefetches. If apps were to fetch network content solely on demand, we would be faced with the opposite problem of automatically prefetching that on-demand content for non-data-conscious users. That reverse problem is *harder* than the one we tackle. In that problem, the solution will need to hunt the app for URLs to fetch ahead of the app requesting it. These URLs may not be readily available, especially when the app constructs the URLs on the fly (e.g. a weather URL with a zip code embedded in it).

Limited programmer effort: By design, we avoid requiring the app developer to change her code. This ease comes at the penalty of limiting our savings. As shown in Figure 10, our conservative design is leaving additional savings on the table. This is because in some programming patterns, it is difficult for static program analysis to ascertain that there is no side effect in a network callback method. This can happen, for example, in dynamic dispatch where the implementation of a polymorphic operation is chosen at runtime not at compile-time. While Procrastinator's static analysis is not robust enough and hence is conservative, dynamic analysis could be used in these cases. However, dynamic analysis would add tremendous complexity to our system, and can impose a significant runtime performance penalty on the user of the app. Alternatively, the app developer could give us 1 bit of information for every network call – does the callback have a side effect? That one boolean can help unlock additional data savings. It requires some developer effort in carefully walking through the code affected by every network transfer.

7. PRIOR WORK

Prefetching is a well-studied problem in many areas of computer systems. Here, we focus on prior work in network prefetching in mobile apps.

The most relevant prior work is Informed Mobile Prefetching [9]. IMP is a system where app developers use an API to specify a method that is capable of fetching a network object, a call to consume the network object, or a call to cancel that prefetch. By providing a hint that a network object may be needed, and then later specifying either the immediate need for it or a cancellation, the underlying system can optimize for data, power and performance goals. Procrastinator and IMP represent two different points in the design space. IMP is a system designed to optimize *prefetching*. On the other hand, Procrastinator never prefetches on its own – it only *delays* fetching of network objects. IMP takes energy consumption into account when making prefetching decisions, while Procrastinator does not. IMP requires deliberate rewriting of app by the developer, and thus, the evaluation of IMP was limited to just two apps. On the other hand, Procrastinator works via binary rewriting, without any input from the developer. As a result, we are able to evaluate Procrastinator on many more apps. IMP includes complex mechanisms for self-learning and self-tuning, while the design of Procrastinator is much simpler. However, learning from past history of procrastination may be beneficial for our system as well. We believe that IMP and Procrastinator offer two design choices that are on extreme ends of the developer effort spectrum. As a result, the system design and techniques that we use are completely different.

This paper is an extended version of a short workshop paper [17]. In this paper, we address a number of deficiencies in that workshop paper. We have made our system significantly more robust by handling more prefetch situations and filling gaps in our system design. This is evidenced by the additional detail in § 3, and our ability to run Procrastinator on many more apps. We have added several elements to our evaluation, including energy consumption, UI automated runs of 140 apps, impact on app code size, and runtime delay.

Outside of network consumption, prior work has modified mobile apps to improve performance and reduce battery consumption. RetroSkeleton [7] is a toolbox to rewrite Android apps without using app source code. While they do not use it to improve network efficiency, they do use it to change some network calls, such as changing all HTTP calls to HTTPS.

Researchers have also looked at automatic static and dynamic analysis of call graphs and data flows in mobile apps for other purposes. TaintDroid [8] dynamically monitors the data flow of apps to find privacy leaks. ADEL [21] 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.

Improving web caching in apps [15] and redundancy elimination of cellular traffic [14] are two additional techniques for reducing data consumption. Our approach is complementary in nature, in that Procrastinator can further reduce data consumption by eliminating some network transfers. Much of Procrastinator's savings are from not fetching unnecessary images, which are typically already compressed in PNG or JPEG formats and would not see much benefit from additional compression. The open-source PageSpeed module [4] for webservers can speed up web browsing and reduce data consumption by optimizing the layout of web pages. It combines and "minifies" resource files, and resizes and re-encodes images. PageSpeed works on web pages, but not on individual web transfers that apps do. Furthermore, Procrastinator applies techniques that PageSpeed does not.

Prior work has also looked at the related problem of cellular signaling overhead and optimizing network traffic burstiness to reduce such overhead and energy consumption. Huang [11] has studied data transfers when the phone's screen is turned off, and proposes being more aggressive in optimizing traffic then. ARO [16] is a system for optimizing the radio behavior of apps to mitigate the ill effects of bursty network usage.

8. SUMMARY

As more cellular operators transition from unlimited data plans to tiered plans, more users will find themselves needing to curb their cellular data consumption. However, many smartphone apps are designed to prefetch network content. We argue that asking an app developer to modify her app is not a winning proposition because it asks for altruism from the app developer to think hard about each network transfer and rewrite code around it. Conversely, automatically deciding whether to do a requested network transfer is hard because it depends on understanding the intent of the app developer. In Procrastinator, we attempt to automatically understand this intent by tying each network transfer to the portion of the UI that is affected. We run into trouble when a network callback method calls a series of other methods, where programming patterns such as reflection can create type instances at run time. In those situations, we cannot statically determine that there will be no ill effect of not running that code. Even with dynamic analysis we may not know without actually running that code. In those situations, we take the conservative approach of allowing the app to proceed, thereby leaving out some data savings for the user.

Nonetheless, on professionally-written apps, Procrastinator has achieved as much as 4X reduction in bytes transferred by the apps for some users. In lab experiments with UI automation, Procrastinator shows 2.5X and higher savings in 20-40 apps. If the user decides to visit off-screen content that we do not prefetch, the penalty is additional latency, which was under 300ms in the median on LTE in our user study. This is only experienced when the user is on cellular and running low on data bytes – otherwise, normal app behavior occurs. Our measured latency is tolerable, and can be seen in our online video [5] where the smartphone was connected over LTE.

Acknowledgments

We thank our shepherd, Ulas Kozat, and the ACM MobiSys 2014 reviewers for their feedback on our work.

9. REFERENCES

[1] Fiddler. `http://fiddler2.com/`.

[2] How to adjust data usage using the Data Sense API. `http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207005.aspx`.

[3] Nokia Lumia 520 phone, as shown on 4 December 2013. `http://www.microsoftstore.com/store/msusa/en_US/pdp/Nokia-Lumia-520-No-Contract-for-ATT/productID.283842800`.

[4] PageSpeed Module. `https://developers.google.com/speed/pagespeed/module`.

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

[6] M. Barnett and M. Fahndrich. Microsoft Research Common Compiler Infrastructure. `http://research.microsoft.com/en-us/projects/cci/`.

[7] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *ACM MobiSys*, 2013.

[8] 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 *USENIX OSDI*, 2010.

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

[10] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *ACM MobiSys*, 2012.

[11] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Screen-Off Traffic Characterization and Optimization in 3G/4G Networks. In *IMC*, 2012.

[12] S. Lidin. Inside Microsoft .NET IL Assembler. In *Microsoft Press*, 2002.

[13] C. Perzold. Microsoft Silverlight Edition: Programming Windows Phone 7. In *Microsoft Press*, 2010.

[14] F. Qian, J. Huang, J. Erman, Z. M. Mao, S. Sen, and O. Spatscheck. How to Reduce Smartphone Traffic Volume by 30%? In *PAM*, 2013.

[15] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Web Caching on Smartphones: Ideal vs. Reality. In *ACM MobiSys*, 2012.

[16] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *ACM MobiSys*, 2011.

[17] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to procrastination and stop prefetching. In *HotNets*, 2013.

[18] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *ACM MobiSys*, 2014.

[19] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *USENIX OSDI*, 2012.

[20] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. In *IEEE Pervasive*, 2009.

[21] 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.