UIWear: Easily Adapting User Interfaces for Wearable Devices

Jian Xu*
Stony Brook University
jianxu1@cs.stonybrook.edu

Qingqing Cao*
Stony Brook University
qicao@cs.stonybrook.edu

Aditya Prakash
Stony Brook University
adiprakash@cs.stonybrook.edu

Aruna Balasubramanian
Stony Brook University
arunab@cs.stonybrook.edu

Donald E. Porter
The University of North Carolina at
Chapel Hill
porter@cs.unc.edu

ABSTRACT

Wearable devices such as smartwatches offer exciting new opportunities for users to interact with their applications. However, the current wearable programming model requires the developer to write a custom *companion app* for each wearable form factor; the companion app extends the smartphone display onto the wearable, relays user interactions from the wearable to the phone, and updates the wearable display as needed. The development effort required to write a companion app is significant and will not scale to an increasing diversity of form factors. This paper argues for a different programming model for wearable devices. The developer writes an application for the smartphone, but only specifies a UI design for the wearable. Our UIWear system abstracts a logical model of the smartphone GUI, re-tailors the GUI for the wearable device based on the specified UI design, and compiles it into a companion app that we call the UICompanion app. We implemented UIWear on Android smartphones, AndroidWear smartwatches, and Sony SmartEyeGlasses. We evaluate 20 developer-written companion apps from the AndroidWear category on Google Play against the UIWear-created UICompanion apps. The lines-of-code required for the developer to specify the UI design in UIWear is an order-of-magnitude smaller compared to the companion app lines-of-code. Further, in most cases, the UICompanion app performed comparably or better than the corresponding companion app both in terms of qualitative metrics, including latency and energy, and quantitative metrics, including look-and-feel.

1 INTRODUCTION

After decades of largely homogeneous personal computing devices, the range of device form factors and user interfaces has widened dramatically. Wearable devices, such as smartwatches and smartglasses, provide user interfaces that decouple the benefits of computer assistance from constantly holding a phone or sitting at a

---

*Primary authors with equal contribution.

computer. For example, a GPS heads-up display helps skiers navigate a difficult slope [11]. The *Spotify* smartwatch app [20] allows the user to control the music easily, freeing the user's hands from constantly holding the phone.

A key bottleneck in leveraging these wearable form factors is the developer effort involved in adapting an application to a wearable device. The dominant model for wearable apps is a *companion app*, that essentially mirrors the smartphone application onto the wearable form factor. The companion app maps a subset of the smartphone app's graphical user interface (GUI), and *re-tailors* the GUI to suit the wearable device. For example, a button on a smartphone might be enlarged and placed on a *card* [7] surface on the smartwatch so that the user can tap anywhere on the watch surface to toggle the button. Our study of the top 100 smartwatch applications in the AndroidWear category [23] on Google Play shows that, for 78% of the apps, the watch app simply mirrors the phone (§2).

For each application and form factor, developers need to write a unique companion app—each requiring thousands of lines of code (§8). As a result, only a small fraction of smartphone apps have companion apps. Moreover, without source code, third-party developers cannot design new companion apps for an existing smartphone app.

In this work, we present a different programming model for wearable devices, where the developer writes the smartphone application *once*, and writes a simple *meta program* to specify how the application should be extended to a given form factor. Our system, UIWear, automatically creates companion apps, that we call *UICompanion* apps, for different form factors based on the meta program. The developer effort in writing the meta program is an order-of-magnitude lower than writing the companion app. UIWear is transparent to the application, does not require source code or changes to the original application, can re-tailor the GUI to suit the wearable interface, and is well-suited for mobile and wearables.

The overarching principle of UIWear is to decouple application management from application design. The human designer (developer or end-user) only specifies what parts of the smartphone GUI are mapped to the wearable and how the GUI is re-tailored, encoded as a meta program.

UIWear's *UI virtualization architecture* manages all other aspects of extending the app to the wearable device. UIWear abstracts a logical model of the GUI and uses this abstraction to re-tailor and render the GUI on the wearable. UIWear also performs all underlying synchronization at the GUI layer. All user interactions on the wearable device are captured by the GUI abstraction and sent to the phone. The UIWear proxy on the phone emulates the

user interaction as though it occurred locally, and the resulting update is mirrored on the wearable device.

One essential contribution of UIWear is identifying the "pinch-point" to extract a logical model of the GUI. Existing works on cross-device display, including RDP [59], and others [25, 27, 44, 60, 62], extract GUI at the hardware frame buffer level. Unfortunately, the frame buffer already composites the UI elements into a bitmap, losing any semantic information that is required for re-tailoring. Instead, UIWear uses the *UI tree* abstraction, which is commonly used across operating systems to represent the GUI as individual UI elements and their relationships [2, 4]. UIWear essentially splices the UI tree, tailors it according to the meta program, and compiles it into the UICompanion app. Although prior approaches [12, 30, 38, 55, 67] have shown how to transparently extract the UI tree without requiring application support, they are only able to extract the UI tree partially. UIWear augments the underlying UI tree mechanism to transparently extract the complete tree.

Another key challenge is in synchronizing at the GUI layer. Smartphones only allow a *single* active GUI at a given time; in fact, the graphics stack of background apps is destroyed. The consequence is that the wearable app can only synchronize as long as the phone app is in the foreground. We observe that, while maintaining the full graphics stack of each background application is power consuming, the logical GUI model (in our case the UI tree) can be kept active even for background apps with minimal effect on power. In UIWear, we modify the operating system to keep the UI tree of a background application active. This simple change allows us to multiplex I/O events to and from background applications with minimal effect on power.

There has been related work both in the systems and the human-computer interaction (HCI) communities for designing cross-device applications. However, related systems either only work in homogenous environments that do not require UI re-tailoring [24–26, 44, 59, 60], or support UI retailoring, but only for specific applications and devices [30, 50, 53, 65, 70]. UIWear combines the advantages of the two approaches.

UIWear is implemented over Android, AndroidWear (for smartwatches), and the Sony SmartEyeglass [17]. The meta program is specified as an XML. We built an easy-to-use overlay visual tool for the designer to mark parts of the GUI that are mapped to the wearable. UIWear then provides XML based templates for the designer to easily write the GUI layout for the wearable device.

We evaluate 20 developer-written companion apps from the AndroidWear category on Google Play. In each case, we create a UICompanion app to mimic the existing companion app functionality. The meta program used to create these UICompanion apps is an order-of-magnitude smaller compared to the AndroidWear companion app. Our evaluation shows that UICompanion apps perform similarly or better for most of the 20 apps in terms of update and response latencies, as well as CPU consumption. Further, UICompanion apps significantly reduce the energy consumption on the watch, in some cases by more than 5 times, by offloading most of the tasks to the more powerful phone. Interestingly, despite offloading the tasks to the phone, UIWear does not increase energy consumption on the phone when compared to companion apps.

We show that UIWear can also be extended to create companion apps on smartglasses. On the Sony SmartEyeglass [17], we create
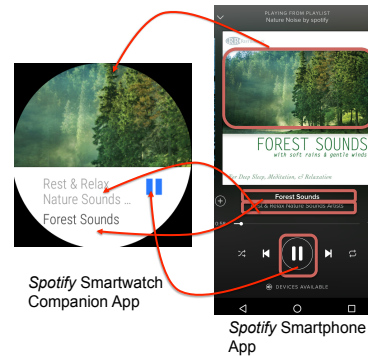

Figure 1: The *Spotify* smartphone and the corresponding smartwatch companion app. A subset of the smartphone GUI (marked within the *red* box) is re-tailored for the watch.

UICompanion apps for two apps, Spotify and MyShoppingList, with less than 30 lines of meta program.

Finally, we conduct a small scale user study to show the effectiveness of UICompanion apps in terms of functionality and look-and-feel. In our study, 20 subjects evaluated five applications using A/B testing techniques. The subjects reported that the UICompanion app was equal or better that the companion version 74% of the times in terms of functionality and 78% of the times in terms of look-and-feel. We also asked 10 of the 20 subjects to write a meta program for two applications. The subjects were able to write the meta program in under 16 minutes.

2 COMPANION APPLICATIONS

Companion apps are a common programming paradigm used by wearable operating systems and vendors including AndroidWear [23], Apple [5], Tizen [18], Pebble [15], and Google glass. A companion app is a GUI projection of the original smartphone app, where parts of the smartphone UI are re-tailored and displayed on the wearable. In most cases, the companion app does not operate independently but synchronizes with the smartphone app.

Figure 1 shows the *Spotify* companion app from the Android-Wear [23] category on Google Play. The companion app exports a view of the Spotify smartphone app to the watch. Events, such as pausing the song, are synchronized with the smartphone.

2.1 AndroidWear Apps

We studied the top 100 applications (as of March 2017) in the AndroidWear category on Google Play to categorize them in Table 1 on a spectrum from (A) completely independent of any smartphone interaction, to (D) Apps whose GUI is a subset of the smartphone GUI, and the wearable device completely mirrors changes on the phone. Between (A) and (D) are two categories: (B) apps that collect local data such as sensors, and (C) apps whose GUI is not a perfect mirror of the phone. Some apps in category (C) allow the two GUIs to drift out-of-sync in order to save energy. Our work focuses on the largest category of apps—category (D) at 78%—which mirror a subset of the GUI. We believe that (C) could likely also be subsumed

by UIWear, and (B) would be possible in future work with some extensions.

We arrived at this categorization by manual inspection of the 100 applications to determine if they are standalone apps. To determine if the application uses local data, we look at permissions, since obtaining local sensor or GPS data requires explicit permission. To check if the app mirrors the smartphone UI, we perform various activities on both the phone and the watch and check for UI updates. Some apps also provide settings to allow users to choose between syncing with the phone or updating locally. Any app with this setting is marked C. The remaining apps, i.e., where the smartphone and the watch UIs mirror each other are marked D. *Watch Faces* are excluded in this study since they only change the watch background.

Category	% Apps	Examples
A	10%	Google Translate, TalkBack
B	6%	Fit, WeChat
C	6%	Outlook, Google Music
D	78%	Spotify, Weather

Table 1: Top 100 smartwatch apps in the AndroidWear category on Google Play divided among four subgroups.

Table 1 shows the percentage of apps and examples in each category. Over 78% of the applications fall under category D. This large number is attributable to a number of factors, including the small energy and CPU budgets on wearable devices, the fact that most users have a smartphone nearby at all times, and a desire to give users a consistent experience and data access on both devices. Even the newer Android 2.0 [3] OS which has about 25 (non watch face) applications on Google Play, only has 7 standalone apps. Although our manual categorization may miss some caveats in app functionality, the trend is clear. We expect that the companion app pattern will remain a common design pattern for wearable applications.

Spotify companion app: To analyze the principal implementation tasks for a companion app in category D, we analyze the Spotify companion app shown in Figure 1. The Spotify smartwatch app is about 10,000 lines of code (LoC), excluding libraries and comments. The app performs the following tasks (with the corresponding lines of code for each task): (1) creates native widgets to display on the wearable GUI and updates the GUI (> 4000 LoC) (2) implements a custom RPC protocol to exchange user input and other application data with the smartphone (> 4000 LoC), (3) creates basic data structures and classes (< 1000 LoC).

The Spotify companion app is a relatively simple adaptation of the smartphone GUI, yet this app is complicated by entangling the wearable GUI design problem with network protocol design to synchronize application state across devices. Our experience is that the Spotify app is representative of other companion apps. The current body of companion apps include one-off implementations of highly-similar management sub-tasks, including positioning GUI widgets on the remote device, synchronizing GUI state with the smartphone, and capturing user input. The goal of UIWear is to decouple GUI design from GUI management.
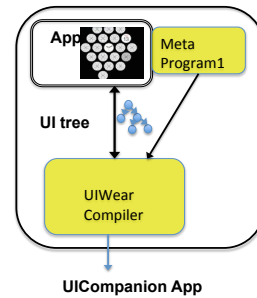
(a) UIWear Compiler and Meta program: UIWear compiles the meta program specification to create the UICompanion app.
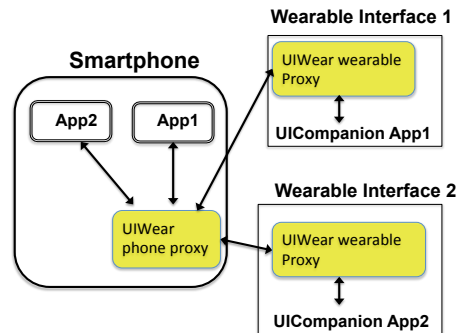
(b) UIWear Runtime: The UICompanion app on the wearable device synchronizes state with the smartphone app to relay user interactions and get UI updates.

Figure 2: UIWear Architecture

3 UIWEAR ARCHITECTURE

UIWear consists of both compile time and runtime components, depicted in Figure 2. At compile time, the human designer writes a meta program and UIWear compiles this into a UICompanion app. The UICompanion app is similar to a developer-written companion app, except that the app is automatically generated by UIWear, based on the design specified by the meta program. At runtime, UIWear synchronizes the application state between the companion app and the phone app, so that a user can access, control, and modify the application from the wearable device.

UIWear meta program: This is the only step of UIWear that requires a human-in-the-loop. Given an application, a developer writes a custom translation of the application UI to the wearable device. The custom translation includes decisions about what parts of the UI are to appear on the wearable device and how the UI is re-tailored. The designer writes a separate meta program for each wearable form factor, such as a smartwatch or a smartglass.

UIWear compiler: The UIWear compiler compiles the meta program to the UICompanion app and ships the app to the wearable device. This compilation step requires that UIWear extracts a logical model of the application GUI and re-tailors the GUI. UIWear uses the *UI tree* as the logical abstraction of the GUI.

A natural point at which to extract the UI tree is at the accessibility and UI automation interface. Current accessibility interfaces
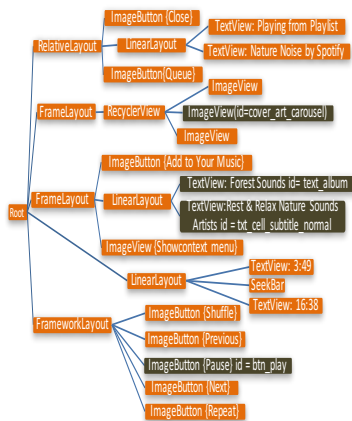
Figure 3: The UI tree corresponding to the Spotify phone GUI shown in Figure 1. The parts marked in *black* show the parts of the UI tree that are chosen by the human designer for display on the wearable device.

are designed only for extracting textual content of the GUI, primarily for applications for the visually impaired that may not require graphical content. UIWear augments this existing technique so that the complete UI tree, including text and visual data are extracted.

UIWear runtime: At run time, UIWear synchronizes the UI states of the wearable app and the corresponding phone app. UIWear's synchronization protocol relays input events from the wearable device to the UIWear phone proxy, which emulates the user input on the phone. The resulting GUI update is relayed to the wearable device using the UI tree abstraction. This synchronization requires no support from the phone application, unlike existing companion apps that need to write a custom remote procedure call interface. One constraint is that, in several smartphone OSes, only a single GUI (of the foreground application) is active at any given time. In UIWear we show how we can multiplex I/O events to and from several applications, including those in the background, at little-to-no marginal cost on the phone.

4 UI TREE EXTRACTION

UIWear uses *UI trees* to represent the logical model of the GUI. The UI tree is effectively a high-level, intermediate representation of the GUI. Both mobile and desktop operating systems use some variation of a UI tree to represent GUI state before it is rendered on the display frame buffer [2, 4, 30].

In Android, the key element required to construct the UI tree is the *view node*. The view node is an intermediate layer residing between the application and the graphics stack (other operating systems have a similar layer). The view node maintains information about each UI element in the application GUI. The view node then calls the graphics stack to composite the UI into a bitmap to render on the screen.

The UI tree is constructed by querying the view node for information about each UI element. Figure 3 shows an example UI tree representation corresponding to the Spotify app shown in Figure 1. Each vertex in the tree is a *UI element* such as a button, a scroll bar, a background image, or a text view. The attributes of these
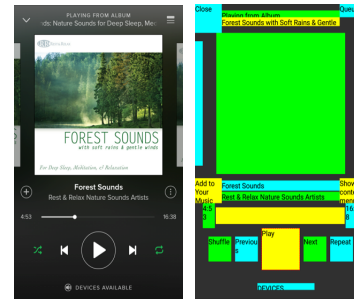

Figure 4: The left shows the original UI and the right shows the UI reconstructed with the UI tree extracted using the accessibility service. Accessibility services are the state-of-the-art for transparently extracting the UI tree, but only extract the textual content.

elements include text, image information, location, and available UI actions, such as whether the element can be *clicked*.

Limitations of current UI tree extraction techniques: The state-of-the-art technique for extracting UI trees transparently, without application support, is through an *accessibility service*. The intended purpose of an accessibility service is to create accessible applications or assistive software for users with visual or other impairments [21]. These interfaces are more general and powerful than just accessibility, and have been used for software testing, UI automation, and other purposes [30, 38, 55, 67].

However, current accessibility APIs are insufficient for UIWear, since they only extract textual information, primarily motivated for helping users with visual impairments. Figure 4 is the reconstructed UI using current accessibility APIs, showing that the UI tree extraction is incomplete. Some tools such as UI Automator Viewer [67] and HierarchyViewer [12] show the complete UI tree either by superimposing the UI tree with the bitmap image, or only allow the UI tree to be accessed from a desktop. Neither of these approaches are suitable for UIWear.
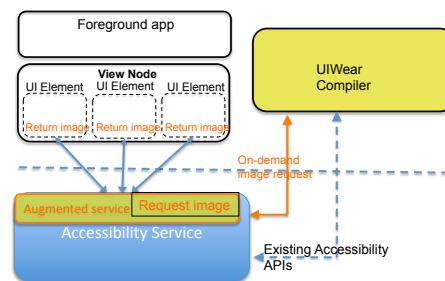

Figure 5: Augmenting the accessibility service to extract the application UI tree. The parts marked in orange show UIWear's augmentation.

UIWear's Augmented Service: UIWear augments the accessibility service to extract the complete UI tree. The default accessibility service works by creating an inter-process communication (IPC) channel between the accessibility service and the view node, which stores the UI tree. Figure 5 shows the UIWear augmented accessibility service. UIWear adds a new query/response API, to request
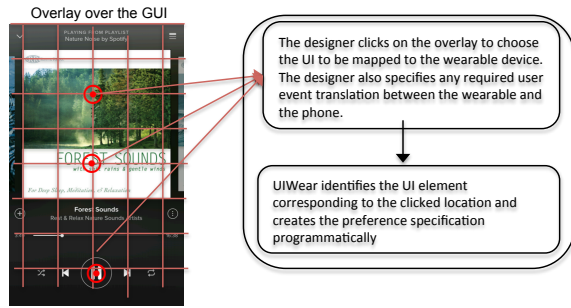
Figure 6: UIWear creates an overlay over the application GUI so that the designer can choose UI elements to map to the wearable using simple clicks.

and receive the image information available at the UI element. To reduce overhead, UIWear requests images on-demand only if the image is going to be used by the UICompanion app.

Beyond the scope of UIWear, a wider range of needs could be met with the augmented accessibility service. For instance, users with lowered vision, but not complete blindness, use screen magnifiers, such as MAGic [35] and ZoomText [22], which enlarge a small portion of the screen in addition to reading GUI contents. These magnifiers must currently resort to other hacks such as hooking the VGA pixel buffers, but could instead use the augmented accessibility service. Similarly, a number of projects have worked on automatically generating text describing images [45, 54]—an enhancement for blind users—and would benefit from the augmentation.

5 META PROGRAM AND COMPILER

The UIWear compiler creates the UICompanion app by combining the meta program and the UI tree. The goal of the meta program specification is to be expressive and yet be simple for a designer to write. The meta program specification consists of two parts: *(i) the preference specification:* containing the subset of UI elements from the original application GUI to be mapped to the wearable. UIWear provides an easy-to-use overlay visual tool for the designer to specify the preference, and *(ii) the wearable layout specification:* specifying how the UI elements on the phone are re-tailored for the wearable UI. We distill a small set of *templates* to make it simpler for designers to specify the wearable layout. Both the preference and the wearable layout specifications are in XML format.

5.1 Preference Specification

The designer creates the preference specification by choosing parts of the GUI to be mapped to the wearable by simply clicking on the objects on the phone UI (illustrated in Figure 6). UIWear combines the location of the click and the UI tree to extract the UI elements to be mapped.

When required, the designer also specifies a translation between the user events on the watch and the phone. For instance, a wrist gesture on the smartwatch is used to scroll the screen, but the same wrist gesture is not meaningful on the phone and needs to be converted to a "scroll" event. When the user event on the wearable input is sent to the phone, the input is translated so that the corresponding action is performed on the phone. In this paper,

we focus on relatively simple translations, but the framework is generic enough to add more complex interactions, such as replacing a textbox with a voice-input class. We are exploring more complex translations as ongoing work.

5.2 Wearable Layout Specification

The wearable layout is used to specify how the phone UI is re-tailored to the wearable device.

Examples. We describe three examples of existing smartwatch companion apps that show the need for re-tailoring the GUI. The *ShoppingList* app on the phone uses a *ListView* to store items in the shopping list. However, the list is not easy to view on a watch. Instead, the corresponding companion app uses the *card* [7] UI design pattern. This pattern groups related information in a container resembling a playing card. Each card contains one shopping item and the user browses the list by swiping through the cards. The *WaterDrinkReminder* app on the smartphone displays the water consumption levels in text. The corresponding companion app transforms text to a progress bar for easy glanceability. Finally, the Spotify companion app, shown in Figure 1, maps four independent UI elements on the phone onto the *card* design pattern. The entire card is *clickable*, and the user can click on any part of the card to play and pause the song.

Wearable layout for Spotify. Figure 7 shows an example wearable layout specification for the Spotify app. The four UI elements from the Spotify phone app (marked in red box in Figure 1) are identified with IDs: backgroundID, titleID, iconID, and textID. On the wearable device, the layout specifies how these four UI elements are laid out using the card design pattern.

The wearable layout adheres to responsive application design [16], where the layout adjusts to different screen sizes and screen shapes. For example, the layout in Figure 7 uses directives such as *match_parent* and *wrap_content* to specify a relative layout according to the parent layout and the screen width respectively. The specification can also include other directives supported by wearable OSes, for example, to adapt the layout for a square screen versus a circular screen [7].

Easing designer effort using templates: To ease designer effort in creating the wearable layout, UIWear provides templates. Most wearable operating systems provide a set of pre-defined UI patterns for designing wearable applications [19]. Developers may also create custom designs. For the 20 AndroidWear applications that we consider for evaluation, only one application used a custom design; the rest used pre-defined templates. For example, the Spotify AndroidWear app uses a pre-defined card template provided by AndroidWear. With the UIWear templates, the designer only needs to fill in the data, such as the ID and position of the UI elements.

By using the template, subjects in our user study without prior experience in wearable programming were able to write the wearable layout specification in under 16 minutes (§8.6).

5.3 UIWear Compiler

The preference file and the wearable layout are combined to create the wearable UI tree, similar to the UI tree shown in Figure 3. UIWear creates the UICompanion app using the wearable UI tree. Creating a UICompanion app involves (1) converting the wearable UI tree

```
<wearable.view.CardScrollView>
    id="@+id/backgroundID"
    layout_width="match_parent"
    layout_height="wrap_content"
    layout_box="bottom">
    <wearable.view.CardFrame>
            <TextView
            id="@+id/titleID"
            layout_width="match_parent"
            layout_height="wrap_content"/>
        <ImageView
            id="@+id/iconID"
            layout_width="30dp"
            layout_height="30dp"/>
        <TextView
            id="@+id/albumID"
            layout_width="match_parent"
            layout_height="wrap_content" />
    </wearable.view.CardFrame>
</wearable.view.CardScrollView>
```

Figure 7: The Spotify wearable layout xml that specifies how the four phone UI elements (marked in the red square in Figure 1) are laid out on the watch. The UI element IDs are backgroundID, titleID, iconID, and albumID.

into a GUI of the proper size to display on the wearable device by detecting the target screen size, and (2) creating appropriate *call backs* to capture user interactions. The new companion app is created *completely* programmatically and does not require manual effort.

UIWear first inflates[1] the UI tree through the corresponding XML. For each UI element, UIWear extracts from the UI tree its image information, textual data, and events associated with it. The image and text information are used to fill the inflated GUI. For each event, for example, a *click* or a *long press*, UIWear creates a callback. The callback simply captures the user interaction, possibly translates the interaction to be meaningful on the phone, and sends the interaction and the UI element ID to the phone.

6 UIWEAR RUNTIME

In the default, non-UIWear companion app model, the application synchronizes its own state across devices using a custom protocol over network sockets. This approach to synchronizing application state requires developers to write a significant amount of error-prone code, as well as reason about and prevent potential concurrency bugs. Instead, UIWear performs an application-independent inter-device synchronization at the UI level. UIWear synchronization is based around the UI tree abstraction.

UIWear phone proxy: Upon changes to the phone GUI (as indicated by an accessibility event), the phone proxy parses the changed UI tree. When applicable, the phone proxy also fetches embedded images associated with certain UI elements in the tree using the augmented accessibility service (Figure 5). UIWear checks if the subset of the UI elements that appear on the wearable device have changed, by comparing them with a previous cached version. If no previous version exists or if there are changes, the proxy ships the *diff* to the wearable device. UIWear also maintains a small cache

---

[1]Inflation is a term used in Android for parsing the XML and turning it into a representation that can be rendered.

at the wearable side so that the same image does not have to be shipped repeatedly.

UIWear wearable proxy: On receiving an update, the wearable proxy forwards the update to the corresponding UICompanion application, which applies this update to its layout and renders the new UI. If the user interacts with the UICompanion app, the wearable proxy captures the interaction and serializes the information to the phone proxy. Example user events include clicks, long press, swiping, and gestures such as wrist movements. The phone proxy then delivers the user event to the application using the accessibility service, as though the event occurred locally on the phone. No change to the application is required. In some cases, the user event needs to be translated from the wearable to the phone, as specified in the meta program (§5). A few user events, such as zoom, are performed locally and not delivered to the phone.

If a user interacts with both the phone and watch interfaces simultaneously and very quickly (or the network is very slow), it is possible to have conflicting updates. In general, the UIWear protocol will serialize events in the order they are received by the application on the phone. We expect that, in the common case, these events will be serialized and processed faster than most humans can input them across devices.

Overcoming smartphone I/O restrictions: A current limitation of most smartphone OSes (including Android, on which we implement UIWear) is that they only allow a single GUI to be active at a given time. The OS shuts down the graphics stack for an application when it is no longer a foreground process. Not maintaining the graphic stack for background apps is a reasonable optimization to save power required for rendering and display. However, the implication of this design is that the GUI synchronization can only work with a foreground app. In order for background apps to be able to receive events from a UICompanion app, and respond to those events, we must keep the UI alive for background apps.

We observe that most of the savings from putting an app in the background are from not *rendering* the composite (pixel-level) display image in the graphics stack. The marginal cost of maintaining the logical UI structure in the view node is relatively low. Therefore, UIWear modifies the OS to disable the graphics stack for background apps, but keeps the view node active. UIWear will only keep the view node active if the app that moves to the background was previously interacting with a wearable device; non-UIWear applications are not affected by this change.

Thus, UIWear can multiplex I/O events to multiple applications, even if these applications are in the background on the phone. §7 discusses additional implementation details, and §8 measures the incremental energy of maintaining the view node for multiple background apps —showing that the costs are indeed negligible.

7 IMPLEMENTATION

We implement UIWear on the Android ecosystem: Android phone OS, AndroidWear smartwatch OS, and the Sony SmartEyeglass that runs over a version of Android [17]. We implemented UIWear on version 6.0 of Android, although there are no version-specific dependencies that would prevent porting UIWear to other versions of Android.

7.1 Compiler Implementation

From the end-user's perspective, an Android application is made up of multiple **activities** or "windows". For example, viewing the playlist is one activity, and viewing the playback controls is another activity. The designer writes a meta program for each application activity to be mapped to the wearable, and UIWear compiles the meta programs and bundles them into a single UICompanion app. In rare cases, an activity may dynamically generate a UI, in which case we need to write a meta program for the dynamic UI. We did not encounter this case in our experiments.

UIWear uses the standard Android toolchain to create the UICompanion application. The application consists of the source file, the AndroidManifest file, the layout, and a set of resources that are created programmatically. We create templates for the meta program using FreeMarker [10].

7.2 UIWear Synchronization Implementation

We implement the UIWear phone proxy as a client to the accessibility service on Android, and the wearable proxy as a service on the wearable device. The phone and wearable proxies together are implemented in 5,300 lines of code.

For each application, the phone proxy keeps track of preference files and the wearable UI tree for each activity of the application. When the user switches between applications or activities on the wearable device, the phone proxy can identify this switch and perform the necessary action on the corresponding UI. The UIWear watch and phone proxy exchange information over the Google Messaging Service (GMS). GMS encapsulates the data and performs the low-level communication tasks, and can use Bluetooth or WiFi.

7.3 I/O Multiplexing Implementation

To multiplex I/O between several UICompanion apps and background apps, UIWear modifies Android to allow the graphics stack to sleep, but keep the intermediate UI representation (i.e., the view node state) active.
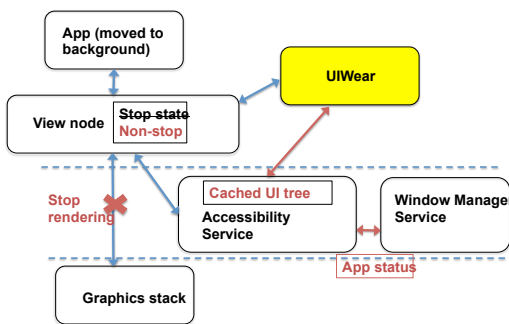

Figure 8: Implementing I/O multiplexing in UIWear: When an application moves to the background, as indicated by the Window manager service, UIWear performs a series of steps to keep the view node and the corresponding accessibility node active. UIWear changes are shown in red.

Figure 8 shows the UIWear implementation. The Window Manager Service indicates if an application is in the background or foreground. Typically, when the app moves to the background, it changes the status of its view node to *stop*, destroys the UI tree maintained by the accessibility service, and makes the graphics stack inactive. Instead, UIWear moves the view node state back to *nonstop* and caches the UI tree of the background application. This simple change makes it possible to multiplex I/O events to background apps.

The Android OS uses a *Low Memory Killer (LMK)* mechanism to kill background applications [1]. There is always a chance that a background app that is supporting a companion application will be killed. However, the LMK mechanism prioritizes background applications that remain inactive for long periods of time. Because the UIWear background applications send and receive events, they are not considered inactive, and are thus less likely to be killed. UIWear applications can also be relaunched by the proxy, if needed.

Our current implementation does not require any OS changes to AndroidWear and Sony SmartEyeglass. Our implementation does change the Android OS, but all changes are in the framework layer. This makes UIWear installation relatively simple as long as the phone is rooted. Further, we believe that there are good technical reasons for mobile OSes to incorporate the augmentations described in UIWear. §4 explains how our augmentations to the accessibility service would benefit other accessibility technologies. Similarly, as wearables and other small computing devices become increasingly ubiquitous, the ability to synchronize updates with multiple (background) apps is fundamentally useful. This functionality has been proposed independently for integrating with automotive displays [43].

7.4 Other Use Cases

While the primary purpose of UIWear is to help designers easily create UICompanion apps, a tech-savvy user can also create their own UICompanion apps even without application source code. We create UICompanion apps for *Yahoo Mail* and *Facebook messenger* (both of which do not have companion apps) in under 25 lines of developer-written XML code. We decompile both the applications using *JADX* [8] and extract the layout XML that specifies the UI. We load the app and use it to specify the preference (Figure 6), and write a layout XML file to re-tailor the phone GUI to the watch.

The Yahoo mail UICompanion app shows the top few emails on the watch and allows the users to read their emails from the watch and the *Facebook Messenger* UICompanion app allows users to read their messages on the watch.

7.5 Limitations and Future work

Wearable apps can perform additional tasks, such as collecting sensor data or running some computation locally; UIWear cannot currently match this functionality. Further, UICompanion apps are completely dependent on having network connectivity to a smartphone, although this problem also extends to most current companion apps. To address these issues, a future extension to UIWear would be to push some amount of computation to the device, such as running small tasks like spell checking or sensor data collection. Other computations, such as a calculator executing the computation on the wearable will require more invasive changes. For this case, we expect that UIWear meta programming model

can be extended to annotate classes that should run remotely. We believe that the general approach to streamlining GUI adaptation will still be useful to these applications.

Further, the current implementation only supports touch and gesture inputs on the wearable, and does not yet support voice. If these additional user events can be encapsulated as a UI widget in the graph, such a translation is relatively straightforward. For instance, one can define an extended text box widget type that includes a voice-to-text synthesis.

UIWear's UI extension does not work for applications such as 3D games that are rendered using OpenGL, since they do not create intermediate UI trees, and instead composite the bitmap directly on the hardware. It is possible that we can intercept some commands at the OpenGL level, but transparently adapting the interface to a large body of graphics-intensive code will be more challenging.

8 EVALUATION

We compare UIWear's UICompanion apps with existing companion apps from the Android Wear category on Google Play using both quantitative measurements (§8.3) and qualitative user studies (§8.6). The evaluation seeks to answer the following questions:

(1) Does the meta programming framework lower development effort to create a wearable app? (§8.1)
(2) How does the end-user latency, CPU utilization, energy usage, and network costs for a UICompanion app compare with the companion app version? (§8.3)
(3) Does maintaining the view tree for multiple background applications affect performance or power? (§8.4)
(4) Can UIWear support heterogeneous form factors? (§8.5)
(5) Do users find the UICompanion app experience comparable to a companion app? How quickly can a new user proficiently write a meta program? (§8.6)

8.1 Applications

We experiment with 20 applications listed in Table 2. We chose companion applications from the top 150 trending apps (as of December 2016). The apps are chosen randomly from among the apps that are in category D (§2), and have diverse popularity, measured in downloads, ranging from 5 thousand to one billion (indicated in parenthesis in the Table).

For each companion app, we write a meta program to create a functionally-identical UICompanion app. Each companion app has between one and nine activities. Our UICompanion apps are written to implement the same number of activities, writing a meta program for each activity. Recall that Android applications encompass one or more activities and the meta program is written at the granularity of an activity (§7).

We are able to create these meta programs without access to source code, as described in §7.4. The LoC for the UICompanion metaprogram shown in Table 2 is for all activities combined, and includes the wearable layout XML, and any lines specified in the preference file to translate user events.

We conservatively estimate the lines of code in each companion app by first decompiling the application using *JADX* [8]. We exclude libraries and blank spaces, and do not count the lines of supporting code on the phone.

App (downloads)	Companion LoC*	Meta prog. LoC
4Sound (100K)	732	21
Anghami (10M)	2,263	79
BandLab (500M)	1,802	8
Counter (100K)	663	15
Endomondo (10M)	6,522	18
Ghostracer (10K)	5,324	95
HydroCoach (1M)	4,022	12
iHeartRadio (50M)	7,926	96
LensGuage (500K)	3,403	24
myTunerFree (5M)	1,535	56
MDPlayer (10K)	1,894	23
MusicPlayer (10M)	540	12
MyShoppingList (50K)	941	42
Parrot (100K)	3,711	9
PlayMusic (1B)	10,851	103
PodcastRepublic (1M)	808	24
Quitter'sCircle(5K)	2,517	17
Spotify (100M)	10,215	47
WaterDrinkReminder(10M)	1,408	18
WorkoutTrainer (10M)	2,265	36

Table 2: 20 companion applications from the AndroidWear category on Google Play used in our evaluations and their downloads in thousands (K), millions (M), and billions (B). The table shows the Lines of Code (LoC) for the smartwatch companion app and UICompanion meta program. *The companion application LoC is a conservative estimate and does not include the LoC at the phone required to support the companion app.*

A UICompanion app requires an order-of-magnitude fewer developer-written code than a comparable companion apps. A meta program ranges from 9–103 lines, and the size is proportional to the complexity of the number of elements and activities in the UI. In total, UIWear represents a significant reduction in effort for wearable app developers.

8.2 Experimental Setup and Methodology

The experiments are performed on two Nexus 5 phones running Android version 6.0.1, and two smartwatches: Sony Smartwatch 3 and Huawei smartwatch running AndroidWear 1.5. All experiments use Bluetooth for communication. In the interest of brevity, we present data from the Huawei smartwatch; the Sony Smartwatch results are very similar.

For fine-grained timing measurement of user events, rendering events, network activities, and CPU measurements, we use recent augmentations [46] to the systrace [69] tool. When needed, we synchronize the clocks on the phone and the watch using ntpd [14]. For apps with more than one activity, we evaluate the activity with the largest number of UI elements. For each activity, we choose all user events on the watch and the phone that generate a GUI update and automate the user events using the adb tool. Each set of user events is repeated 10 times and we report the average and standard deviation.
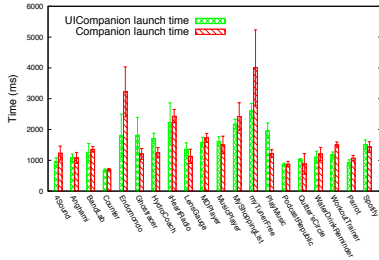
Figure 9: Comparing launch times, the time for the watch GUI to be rendered when it is first launched. The launch time is typically longer compared to response and update times because all images need to be loaded on the watch.
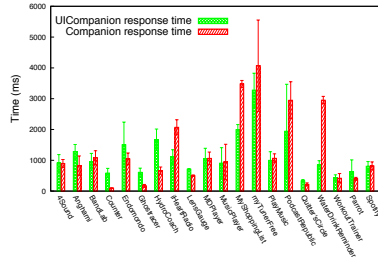

Figure 10: Comparing response times, the time taken for the watch GUI to update in response to a watch user event. The response time requires a round trip, to deliver the user event to the phone and then update the corresponding GUI.
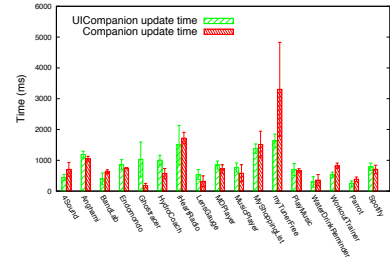

Figure 11: Comparing update times, the time to update the watch GUI in response to an update on the phone. The update times are shorter compared to the response times because the phone only needs to send diffs.

For power measurements on the phone, we use *BattOR*, a small, noninvasive hardware device that interposes between a device and its battery [63]. BattOR collects power measurements on the order of tens of microseconds and is known for high accuracy. Importantly, BattOR can measure power for phones without removable batteries, unlike the more popular Monsoon Power Monitor [13]. We also performed measurements on a Galaxy Nexus phone with the Monsoon Power Monitor, and our results were similar for both monitors. The energy consumption is measured over a period of 1 minute with 10 repeated GUI changes and user interactions. On the smartwatch, we use the dumpsys battery status tool [9]. In the default case, the phone application is in the foreground. §8.4 measures costs when the phone app is in the background.

8.3 Quantitative Comparisons

Although UIWear is not primarily designed to improve performance or resource consumption, UIWear apps perform comparably to (and in some cases even better than) hand-written companion apps.

Latency:

We compare AndroidWear companion and UIWear's UICompanion apps, measuring: (1) launch time, (2) update time, and (3) response time. Launch time is the time from starting an app until all text and image data are fetched from the phone. After initial launch, both the companion and UICompanion apps only need to send *diffs*. The update time is the time for the GUI update on the phone to be reflected on the watch. The response time is the time between a user event at the watch and the corresponding GUI update on the watch. The response time is typically larger than the update time because the response time requires a round trip; the user event is first shipped from the watch to the phone, and the corresponding GUI update on the phone is mirrored back on the watch.

Launch time: In most cases (16/20), the average launch times of companion and UICompanion apps are similar, shown in Figure 9. For three applications, *HydroCoach*, *Ghostracer*, and *PlayMusic*, UICompanion is 30% slower. We discuss why the *HydroCoach* and *Ghostracer* UICompanion apps perform poorly in the next section. With *PlayMusic*, our hypothesis is that the application is heavily optimized for the launch time due to high popularity of the app. In

contrast, *MyTunerFree* and *Endomondo* companion apps are nearly 50% slower than UIWear. In case of *MyTunerFree*, the companion app did not perform well in general.

Response times: For about half of the cases (9/20), UIWear and the AndroidWear companion apps have comparable response times; for about a quarter, UIWear has faster responses, and for the other quarter the hand-written companion app has better response time. The average response times are shown in Figure 10. In general, the apps with comparable response times have comparable network costs (Figure 14).

For five apps, the UIWear response time is much better compared to the AndroidWear version, with an average reduction of 50%. Commiserate with the high response times, the companion app version for these five apps consume more power on the smartwatch compared to UIWear (Figure 15).

On the other hand, apps that can handle some updates locally tend to out-perform UIWear. *Ghostracer* and *Counter* companion apps have lower response times since they update the smartwatch GUI based on local computation (corroborated by network usage in Figure 14). We will investigate how to push modest computation to the watch to improve UIWear performance for such cases. The *Hydrocouch* application requires a lot of items to be fetched from the phone: UIWear extracts each item from the UI tree, whereas the companion app gets the items from the app. This results in higher response time for the *Hydrocouch* UICompanion app.

Update times: For 14 of 20 apps, the average update times were similar for the companion and UICompanion apps, listed in Figure 11. Three companion apps, *Counter*, *PodcastRepublic*, and *QuitterCircle* simply did not update the watch GUI in sync with the phone, so we were not able to compare them with UIWear. As before, the *myTunerFree* companion app is 50% slower than the UIWear version, and the *Ghostracer* and *Hydrocouch* companion apps are faster.

To understand where most of the time is spent, Figure 12 shows the breakdown of the update times for five UICompanion apps. We do not have the source code for the companion apps, and therefore cannot compute such a break down. The network contributes 50% to 80% of the total time. The next biggest bottleneck is the UIWear processing on the phone, to parse the UI tree and create diffs. Rendering contributes little to the overall latency.
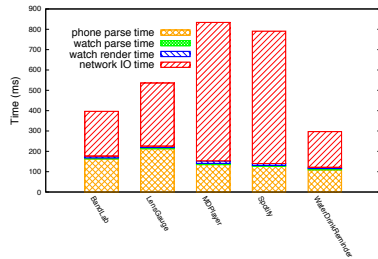
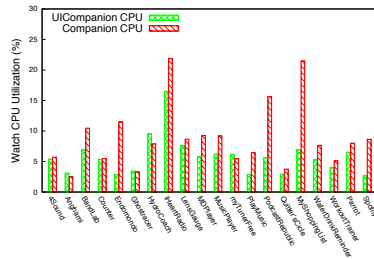**Figure 12: Breaking down the response times latency.**

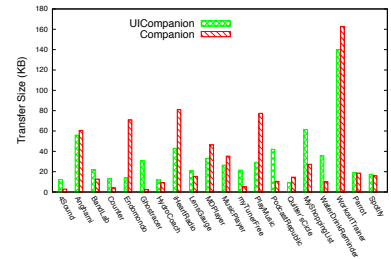**Figure 13: Comparing CPU consumption on the watch.**

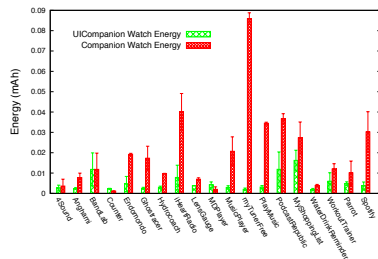**Figure 14: Comparing network data transfer from phone to watch.**

Figure 15: Comparing energy consumption on the watch UICompanion offloading more computation to the phone, reducing energy.
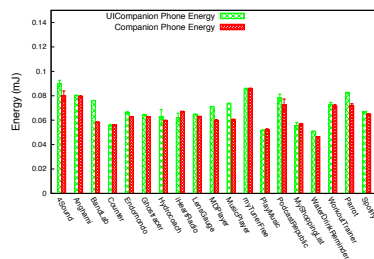

Figure 16: Comparing energy consumption on the phone. UICompanion and the companion counterpart consume have comparable energy consumptions.
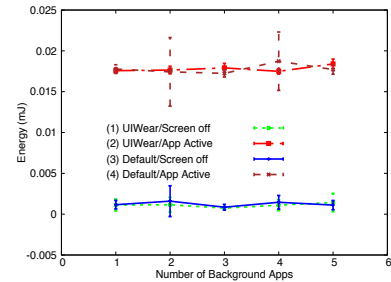

Figure 17: Comparing energy under UI-Wear and under default behavior where the view node is destroyed.

CPU, Energy, and Network Resources

CPU usage: Figure 13 shows that watch CPU consumption for both the UICompanion and the companion apps. For 3 apps, UICompanion app reduces CPU consumption by over 3 times by offloading most of the computation to the phone.

On the phone, the CPU consumption of UICompanion and companion apps is comparable across all 20 apps (2% lower for UICompanion on average). We omit a detailed listing in the interest of brevity.

Energy consumption: UICompanion apps consume *significantly* less energy on the watch than companion apps, depicted in Figure 15. In the case of *myTunerFree* app, UICompanion reduces energy by *20 times*. UIWear essentially shifts processing to the phone, reducing energy usage on the watch.

Even though work is offloaded to the phone, UICompanion apps generally do not increase phone energy consumption, shown in Figure 16. In the worst cases, 4 apps increase phone energy consumption by 3%. We believe this is because the phone is more powerful and the incremental costs of performing the additional work on the phone is lower.

Network usage: Figure 14 shows the amount of data transferred from the phone to the watch. For 9 of the 20 apps, the amount of data transferred by UICompanion and companion apps are similar. In other words, the developers of these 9 apps are also writing companion apps as GUI subsets of the original app, and synchronizing image and other data between the wearable and phone app at run

time. UIWear simply streamlines this process to reduce developer effort. For 5 of the 20 apps, the companion app transfers a large amount of data to the watch; without source code, we are unable to determine the reason for this data transfer. The companion apps that perform more local computation (e.g., *Ghosttracer*, and *Counter*) require less data to be transferred.

8.4 Effect of Multiple Background Applications

One potential concern about UIWear is the incremental cost of having multiple active apps in the background on the phone, responding to I/O from the watch. Figure 17 shows the energy consumption as the number of background applications increases from one to five. We measure this under four conditions: (1) the view nodes are active, but the screen is off, (2) the view nodes are active, and an application is running in the foreground, (3) the view nodes are inactive and the screen is off, and (4) the view nodes are inactive with a foreground app. Cases (1) and (3) represent the UIWear design, and (2) and (4) are Android defaults.

Figure 17 shows that the energy cost of keeping view nodes active with the screen off (1) is comparable to making them inactive (3), and that having additional active view nodes running (2) along side a foreground application is comparable to having no background view nodes (4). In the interest of brevity, we omit performance figures; in summary, there is no difference in performance of the foreground application as the number of background apps increases. There was also no difference in performance on the watch whether the phone app is in the background or foreground.

Metric	UICompanion better or equal	AndroidWear better
Functionality	78%	22%
Look and feel	74%	26%

Table 3: Qualitative comparison of UIWear's UICompanion app and the AndroidWear playstore companion app in terms of functionality and look-and-feel. The results show the percentages across 20 subjects and 5 applications.

8.5 Sony SmartEyeglass

We demonstrate the ability for UIWear to support multiple form factors by also creating UICompanion apps for the smartglass form factor for two applications: Spotify and MyShoppingList. For both applications, the UICompanion app required less than 30 lines of meta program.

On the Spotify UICompanion app on the glass, users can pause or play the song. On the MyShoppingList app, users can delete items. The response time is *74ms* for the Spotify application and *24ms* for the MyShoppingList application, averaged over 10 runs. Since the Sony SmartEyeglass ecosystem does not have existing companion applications, we have no good baseline for comparison. This case study does demonstrate that UIWear can accelerate development of companion apps for a new platform.

8.6 User Study

We conducted an IRB-approved user study to (i) qualitatively compare the AndroidWear playstore companion app and the UIWear generated UICompanion app, and (ii) measure the time taken for a designer to write a meta program specification for a given application.

We recruited twenty subjects from the University: four undergraduate and sixteen graduate students. 30% of the students were female and 70% were male, 30% of the students were from outside the computer science department and the remaining were computer science students.

Qualitative comparisons: We use two qualitative measures for comparison: *look-and-feel* and *functionality*. The experiments are conducted on the following apps: Counter, LensGauge, MusicPlayer, MyShoppingList and Spotify. For each application, we let the subjects interact with the two versions of the app for one minute each. We use A/B testing and record the subjects responses in terms of whether they preferred A, B, or if the two versions were equal. The results of the comparison are shown in Table 3. Subjects found UICompanion version to be equivalent or better than the companion version 78% of the time in terms of functionality and 74% of the time in terms of look-and-feel.

We studied the cases when UICompanion apps performed worse than the AndroidWear companion apps in terms of look-and-feel and functionality. We find that this was likely because the UICompanion app did not port the progress-bar widget and did not support animations. The subjects found the animation and the widgets more appealing. We believe these issues can be addressed in a more fully-featured version of UIWear.

Estimating designer effort: We asked ten subjects to write a meta program to mimic two companion applications: Spotify and Music Player. Before the start of the study, we gave all subjects a ten minute introduction to the procedure. We provide the template for both applications and the preference file as described in §5.

The subjects had to first identify which UI element each ID in the preference file corresponded to and create the layout XML to specify the location and size of each UI element. The subjects were able to view the visual layout. *Each subject created a working meta program in under 16 minutes per application.* Of the ten, five were from the Computer Science department and five were from other departments. Six subjects considered themselves strong programmers and the other four rated themselves as novice programmers.

These results indicate that, from the end-user's perspective, UIWear can generate apps that are competitive with hand-written companion apps. Moreover, the meta programming process is accessible even to novice programmers.

9 RELATED WORK

Broadly speaking, related *systems* research on extending UIs to remote devices do not re-tailor the UI for the remote screen [24, 26, 34, 44, 59–61, 68]. As a result, these works are largely applicable only to homogenous settings where no UI retailoring is needed. Existing *HCI* work on designing cross-device UI focus on (1) designing UIs that are consistent across devices and (2) authoring tools for cross-device UI design. However, these HCI techniques cannot be applied to off-the-shelf applications and they do not solve the systems challenges of extracting and synchronizing UI elements. UIWear fills this void by addressing both the HCI and the systems challenges in extending UI across devices.

9.1 Systems Approaches

Supporting Remote Access

A number of protocols and systems have been designed for remote access to a server or desktop, including RDP [59], VNC [60], ICA [39], PCoIP [66] and many others [25, 27, 44, 62]. In general, these protocols simply relay the hardware framebuffer contents, as well as mouse and keyboard events, between systems. Research proposals have also extended remote access protocols with some semantic information, to cope with high resolution graphics [34], lower-bandwidth remote access connections [27], or to modify gestures [56]. These approaches do not support GUI re-tailoring to suit small form factor devices.

The main remote access protocol that uses semantic objects is X [61]; to our knowledge, X has not been previously used for adapting GUIs to heterogeneous and small form factors. In fact, there are no implementations of an *X client* on phones, although some smartphones do support an X server. The UIWear design is inspired by the X protocol, but is designed specifically for small form factor devices.

Porting Applictations

A number of solutions port applications from one platform to another, that includes porting the UI. A subset of related work in this category [28, 33, 41, 42, 57] port screen access through a virtual frame buffer, which, as stated above, does not solve the problems addressed by UIWear.

Flux [68] ports UI across devices *without* transferring the screen buffer, and instead migrates the application from one device to another. Flux does account for device heterogeneity and can run a smartphone application on a more powerful tablet and vice versa. Flux assumes that architectures of the devices have some differences, but largely provide similar functionality. However, wearable devices only support a small subset of functionalities of a smartphone, making app migration from a smartphone to a wearable device infeasible.

Cider [24] and Apportable [26] port phone applications that are designed for one OS to another. For instance, Cider [24] designs core libraries and compile-time code adaptation to support applications designed for iOS to run on Android. However, similar to the app migration above [68], these systems cannot be extended to highly heterogenous devices.

Most similar to UIWear is Sinter [30], which also mines a UI tree using accessibility interfaces. The goal of Sinter is to make accessibility tools, such as screen readers, work across different OSes and on remote or virtual desktops. Sinter is not sufficient to create a companion app, as it cannot extract or manipulate image or other visual data, does not address heterogeneity in form factors, and does not work when the smartphone application is no longer in the foreground.

9.2 HCI Approaches

Cross-Device UI Authoring tools

UIWear is inspired by the pioneering work in the model-based user interface design paradigm that allows developers to specify the UI using higher-level abstractions, rather than programming a specific UI layout [32, 32, 58, 64].

While the model-based design makes authoring UIs simpler, an abstraction layer has to specified for each concrete device, OS, and UI look-and-feel. As a result, recently, most works use a hybrid method that simplifies some elements of cross-device UI design, but ultimately maintains a WYSIWYG approach for each device type, requiring some manual effort per device type. Gummy [48], PageTailor [29], WinCuts [65] and Jelly [47] start off from a given UI and allow developers to choose pieces of this UI to be reused on a different device. Highlight [53] uses a middleware proxy to compile HTML from PCs to mobile phones rather than requiring a UI abstraction.

However, these projects do not solve the problems addressed in UIWear. While Gummy [48] and Jelly [47] provide developers a platform to *design* UIs for each device type, they do not automatically create the companion application. Specifically, these works do not address the systems challenges of extracting, transferring, and updating the UI.

On the other hand, WinCuts [65], PageTailor [29] and Highlight [53] work only for Web pages. In general, cross-device Web page authoring is a better-studied problem, in part because Web page compilers and Web browsers are more standardized across devices than the runtime environment for mobile applications. For example, recent work on *Responsive UI* platforms [6] allow developers to write a single Web page that can be modified for different screens. However, the ideas used for cross-device Web page designs do not translate to other mobile applications.

Co-designing cross-device UI

UI co-design research focuses on building platforms and tools for developers to simultaneously design UIs for different devices for consistent look-and-feel. Nguyen et al. [52] provides a platform to create a fully-abstracted UI that compiles to concrete UIs on different devices. Weave [31] uses a Javascript-based scripting framework for a developer to concurrently visualize UI on multiple devices. WatchConnect [40] is a toolkit and an emulation platform for users to easily design smartwatch UI. Ghiani *et al.* [36] lets a developer annotate an abstract UI and select which segments of the UI will be rendered on different devices

All of these works make it easy to design cross-device UIs, but they only work for custom applications that are designed using the tool. They do not work for off-the-shelf applications and do not handle any of the systems challenges of sharing the UI.

Distributing the UI across devices

Since wearable computers became mainstream, UI researchers are creating interactions that involve multiple devices working at the same time. In effect, a single application UI is split into different parts and shown on different device screens for the best viewing experience for the user.

The XDBrowser system [50, 51] allows web page authors to distribute the UI across the smartphone, smartwatch, and the PC. Recently, the XDBrowser system was extended for email clients [49]. Similarly, the Panelrama system [70] uses a novel annotation and distribution technique. The developer annotates the Web page and Panelrama automatically renders different parts of the Web page on different devices based on the annotation.

However, existing systems focus on a single application, most commonly Web pages. They do not work for diverse, off-the-shelf, applications. The Conductor [37] system designs new interaction methods for users to migrate between a smartphone and a wearable screen. This work makes it easier to move between UIs, but does not provide techniques to share the UI itself.

10 CONCLUSION

In the near future, we expect to see an increasing abundance of smaller, network-connected devices with diverse form factors and limited computing power. UIWear can dramatically simplify the development effort to export user interfaces to this broad array of devices. The developer writes an application for the smartphone, and UIWear extends the application to any wearable device with minimal effort from the developer, without sacrificing performance or power. UIWear contributes a new design point that decouples the concerns of application logic and UI design.

REFERENCES

[1] Android Internals: A Confectioners Cookbook. http://newandroidbook.com/index.php.

[2] Android View Hierarchy. http://developer.android.com/guide/topics/ui/overview.html.

[3] Androidwear 2.0. http://www.wareable.com/android-wear/.

[4] Apple View Hierarchy. https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/View%20Hierarchy.html.

[5] Apple watchos. http://www.apple.com/watchos/.

[6] Bootstrap. http://getbootstrap.com/.

[7] Card layout. https://developer.android.com/training/wearables/ui/cards.html.

[8] Dex to Java decompiler. https://github.com/skylot/jadx.

[9] Dumpsys battery status. https://source.android.com/devices/tech/power/batterystats.html.

[10] Freemarker. http://freemarker.org/.

[11] GPS Heads Up Display. http://www.reconinstruments.com/products/snow2/.

[12] Hierarchy Viewer. http://developer.android.com/tools/help/hierarchy-viewer.html.

[13] Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[14] ntpd. http://doc.ntp.org/4.1.0/ntpd.htm.

[15] Pebble. https://www.pebble.com/.

[16] Responsive ui design. https://medium.com/google-developers/building-a-responsive-ui-in-android-7dc7e4efcbb3#.bi9jk1rdy.

[17] Sony SmartEyeglass SDK. https://developer.sony.com/develop/wearables/smarteyeglass-sdk/.

[18] Tizen. https://www.tizen.org/.

[19] Ui patterns for wearables. https://developer.android.com/design/wear/patterns.html.

[20] Wear Spotify for Android Wear. https://play.google.com/store/apps/details?id=wearablesoftware.wearspotifyplayer&hl=en.

[21] Android Accessibility Service. http://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html.

[22] Ai Squared. Zoomtext magnifier. http://www.aisquared.com/zoomtext/more/zoomtext_magnifier.

[23] Google Android Wear Market. https://play.google.com/store/apps/category/ANDROID_WEAR?hl=en.

[24] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native execution of ios apps on android. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 367–382, 2014.

[25] Apple. Airplay—play content from ios devices on appletv. https://www.apple.com/airplay/.

[26] Apportable. http://www.apportable.com/.

[27] R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: A virtual display architecture for thin-client computing. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2005.

[28] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[29] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara. Pagetailor: reusable end-user customization for the mobile web. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 16–29. ACM, 2007.

[30] S. M. Billah, D. E. Porter, and I. V. Ramakrishnan. Sinter: Low-bandwidth remote access for the visually-impaired. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[31] P.-Y. P. Chi and Y. Li. Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3923–3932. ACM, 2015.

[32] K. Coninx, K. Luyten, C. Vandervelpen, J. Van den Bergh, and B. Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In *International Conference on Mobile Human-Computer Interaction*, pages 256–270. Springer, 2003.

[33] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[34] C. D. Estes and K. Mayer-Patel. Moving beyond the framebuffer. In *Proceedings of the 21st International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '11, pages 93–98, New York, NY, USA, 2011. ACM.

[35] Freedom Scientific. Magic screen magnification software. http://www.freedomscientific.com/products/lv/magic-bl-product-page.asp.

[36] G. Ghiani, M. Manca, and F. Paternò. Authoring context-dependent cross-device user interfaces based on trigger/action rules. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, pages 313–322. ACM, 2015.

[37] P. Hamilton and D. J. Wigdor. Conductor: Enabling and understanding cross-device interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2773–2782, New York, NY, USA, 2014. ACM.

[38] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[39] J. Harder and J. Maynard. Technical deep dive: Ica protocol and acceleration. http://s3.amazonaws.com/legacy.icmp/additional/ica_acceleration_0709a.pdf.

[40] S. Houben and N. Marquardt. Watchconnect: A toolkit for prototyping smartwatch-centric cross-device applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1247–1256. ACM, 2015.

[41] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 529–546, 2013.

[42] J. Howell, B. Parno, and J. R. Douceur. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the USENIX Annual Technical Conference*, pages 321–332, 2013.

[43] Intel. Intel atom e3800 processor series: Android multi-display features in automobiles. http://www.intel.in/content/www/in/en/embedded/products/bay-trail/atom-e3800-android-display-vehicles-paper.html, 2014.

[44] J. Kim, R. A. Baratto, and J. Nieh. pTHINC: A Thin-client Architecture for Mobile Wireless Web. In *Proceedings of the 15th International Conference on World Wide Web (WWW)*, pages 143–152, 2006.

[45] G. Kulkarni, V. Premraj, V. Ordonez, S. Dhar, S. Li, Y. Choi, A. C. Berg, and T. L. Berg. Babytalk: Understanding and generating simple image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(12):2891–2903, Dec. 2013.

[46] R. Liu and F. X. Lin. Understanding the characteristics of android wear os. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 151–164, New York, NY, USA, 2016. ACM.

[47] J. Meskens, K. Luyten, and K. Coninx. Jelly: A multi-device design environment for managing consistency across devices. In *Proceedings of the International Conference on Advanced Visual Interfaces*, pages 289–296. ACM, 2010.

[48] J. Meskens, J. Vermeulen, K. Luyten, and K. Coninx. Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In *Proceedings of the working conference on Advanced visual interfaces*, pages 233–240. ACM, 2008.

[49] M. Nebeling. Xdbrowser 2.0: Semi-automatic generation of cross-device interfaces. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4574–4584, New York, NY, USA, 2017. ACM.

[50] M. Nebeling and A. K. Dey. Xdbrowser: User-defined cross-device web page designs. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5494–5505. ACM, 2016.

[51] M. Nebeling, T. Mintsi, M. Husmann, and M. Norrie. Interactive development of cross-device user interfaces. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2793–2802. ACM, 2014.

[52] T.-D. Nguyen, J. Vanderdonckt, and A. Seffah. Generative patterns for designing multiple user interfaces. In *3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems MobileSoft'2016*, 2016.

[53] J. Nichols, Z. Hua, and J. Barton. Highlight: a system for creating and deploying mobile web applications. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 249–258. ACM, 2008.

[54] V. Ordonez, W. Liu, J. Deng, Y. Choi, A. C. Berg, and T. L. Berg. Learning to name objects. *Commun. ACM*, 59(3):108–115, Feb. 2016.

[55] E. F. Oriana Riva, Suman Nath. Appstract: On-the-fly app content semantics with better privacy. In *ACM Mobicom*. ACM, July 2016.

[56] Parallels. Parallels access. http://www.parallels.com/products/access/.

[57] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, 2011.

[58] A. R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4):40–47, 1997.

[59] Remote Desktop Protocol. https://en.wikipedia.org/wiki/Remote_Desktop_Protocol.

[60] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33–38, Jan 1998.

[61] R. W. Scheifler and J. Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, Apr. 1986.

[62] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of slim: A stateless, thin-client architecture. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 32–47, 1999.

[63] A. Schulman, T. Thapliyal, S. Katti, N. Spring, D. Levin, and P. Dutta. Battor: Plug-and-debug energy debugging for applications on smartphones and laptops. Technical report, Stanford University, 2015.

[64] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools: the mastermind approach. In *Engineering for Human-Computer Interaction*, pages 120–150. Springer, 1996.

[65] D. S. Tan, B. Meyers, and M. Czerwinski. Wincuts: Manipulating arbitrary window regions for more effective use of screen space. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 1525–1528, New York, NY, USA, 2004. ACM.

[66] Teradici. Pc-over-ip technology explained. http://www.teradici.com/pcoip-technology.

[67] UIAutomater Viewer. http://www.bdtool.net/third/android-doc/web-docs/tools/testing/testing_ui.html.

[68] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: Multi-surface computing in android. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 24:1–24:17, New York, NY, USA, 2015. ACM.

[69] Systrace. https://developer.android.com/studio/profile/systrace-commandline.html.

[70] J. Yang and D. Wigdor. Panelrama: Enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2783–2792, New York, NY, USA, 2014. ACM.