Sensor Virtualization for Efficient Sharing of Mobile and Wearable Sensors

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

Arani Bhattacharya
IIIT-Delhi
arani@iiitd.ac.in

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

Donald E. Porter
UNC Chapel Hill
porter@cs.unc.edu

ABSTRACT

Users are surrounded by sensors that are available through various devices beyond their smartphones. However, these sensors are not fully utilized by current end-user applications. A key reason sensor use is so limited is that application developers must exactly identify how the sensor data can be used by smartphone apps. To mitigate this problem, we present SenseWear, a sensor-sharing platform that extends the functionality of a smartphone to use remote sensors with limited additional developer effort. Sensor sharing has several uses, including augmenting the hardware in smartphones, creating new gestural interactions with smartphone applications, and improving application's Quality of Experience via higher-quality sensors from other devices, such as wearables. We developed and present six use cases that use remote sensors in various smartphone applications. Each extension requires adding fewer than 20 lines of code on average. Furthermore, using remote sensors did not introduce a perceptible increase in latency, and creates more convenient interaction options for smartphone apps.

CCS CONCEPTS

• **Human-centered computing** → **Smartphones**.

KEYWORDS

wearable, sensor, mobile system, usability, gesture, development

ACM Reference Format:
Jian Xu, Arani Bhattacharya, Aruna Balasubramanian, and Donald E. Porter. 2021. Sensor Virtualization for Efficient Sharing of Mobile and Wearable Sensors. In *The 3rd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things (AIChallengeIoT 21), November 15–17, 2021, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3485730.3493451

1 INTRODUCTION

Users today are surrounded by a large and growing number of sensors from various devices other than their smartphones. Specifically, wearable devices are equipped with new sensors including ambient light sensors, heartbeat rate sensors, and near-field communication (NFC). Because wearable sensors are physically attached to users, they can better measure the physical characteristics of users and their environments.

However, end-user applications do not fully utilize these sensors. Despite wearable devices being available for nearly a decade, both stand-alone wearable apps or smartphone apps that leverage wearable sensors are few and far between. From our survey of the top 250 most popular Android Wear apps from the Google Play marketplace, we find that only 16% of apps utilize wearable sensors, and primarily for straightforward use cases like activity tracking and map navigation. Of these 16% of apps, nearly half are standalone watchface apps that do not have any connection to smartphone apps but simply display users' health data on the watch.

One likely reason for this dearth of smartphone apps that leverage wearable sensors is that it is difficult for developers to write code that both uses current sensors and is portable to new sensors. Apps that use wearable sensors require complex code for tasks such as processing sensor data, data communication across devices, and sensor customization across heterogeneous devices [11]. None of those tasks are trivial; for instance, significant background knowledge on signal processing is required to write concise and efficient sensor code [25]. Data communication among devices brings along the complexity of writing a distributed system [35]. Designing an application that uses sensors of another device thus requires hundreds of lines of code [25]. Prior work, such as Mobile Plus [29] and M2 [11], utilize sensor data from one device on another, but they still depend on developers to programmatically specify the type of interaction, such as swiping, tapping the screen. In short, developers struggle with the complexity of using these readily available sensors, and users miss the opportunity for more efficient and customized sensing in their smartphone apps.

To mitigate this problem, we design SenseWear, a sensor sharing platform that facilitates the use of sensor data from wearables, or any other sensor device, by smartphone applications. SenseWear decouples the sensing code from the application logic, creating a modular boundary for plugging in sensor data from different devices. With only a few lines of code, SenseWear can (a) extend a sensor application beyond the hardware on a given smartphone, and (b) improve the user experience on existing apps by transparently and opportunistically using higher-precision sensors available on wearable devices. We discuss these use cases in detail in §2.

SenseWear enables the use of wearable devices as follows:

AIChallengeIoT 21, November 15–17, 2021, Coimbra, Portugal

ACM ISBN 978-1-4503-9097-2/21/11…$15.00
https://doi.org/10.1145/3485730.3493451

Minimize programming effort in sensor mapping: SenseWear significantly reduces the programming effort needed to port wearable sensor data to a smartphone. SenseWear is a platform that easily customizes wearable sensor data so that those sensor data can be seamlessly utilized by smartphone apps. SenseWear contributes a metaprogram framework that abstracts the essential task of mapping sensor input across devices. To extend the smartphone app to leverage remote sensors, the developer need only write a simple metaprogram. The metaprogram language only exposes a few necessary APIs and specification choices for developers, without requiring them to implement details such as inter-device communication or sensor mapping algorithm for calibration across devices. The metaprogram is used by the developer to choose one among the multiple possible types of interaction possible between a remote and a local sensor, either with direct mapping or a semantic mapping. A typical metaprogram in SenseWear is fewer than 20 lines-of-code. SenseWear takes care of the rest of the sensor mapping process by correctly and transparently "plumbing" the data into the application.

Automatically Handling Sensor Heterogeneity: The key challenge developers face is the high degree of heterogeneity across sensors. Specifically, devices can vary widely in sensor specification, orientation, and position, which means that to use a remote sensor, one has to map remote sensor data to the local device. SenseWear uses linear regression to simply calibrate the IMU sensors of the smartphones with that of the wearable device to emulate the type of movement seen by the smartphone. The application developer still writes a monolithic sensor application as if it were designed to only run on a smartphone.

Evaluation: We implement SenseWear in Android OS 7.1.2 and Android Wear 2.0 smartwatch OS. We demonstrate the utility of SenseWear by creating eight SenseWear-enabled apps that showcase the usefulness of leveraging remote sensors. These apps include (1) Two mobile gaming apps where the smartwatch accelerometer is used to control the game instead of tilting the smartphone, (2) Two apps whose haptic feedback is relayed to the smartwatch from the smartphone so that the user gets feedback on a device closer to her, (3) Two apps that allow turning off a smartphone screen by controlling the smartwatch.

In each case, the smartphone application is not re-compiled. Instead, remote sensors are replaced via a small metaprogram (each app takes fewer than 20 lines of code). Our evaluation shows that in all eight cases, the remote sensor latency is lower than what a human can perceive. Furthermore, we show that wearable sensors can be mapped to the smartphone's use with over 90% accuracy.

2 USE CASES

SenseWear aims to enable the development of sensor-based apps for wearables by providing the following benefits.

Overcoming Hardware Limits New types of sensors can unlock new families of applications, but it is unlikely that every smartphone has all kinds of sensors onboard. For example, mobile payment apps require an NFC sensor, but a third of the new phones shipped in 2018 [10] are not equipped with NFC sensors. SenseWear bridges this gap, making it easy to "extend" the sensing hardware of a phone by leveraging a wearable NFC sensor, thereby enabling payment apps on a phone where it would otherwise not run.

Improving Quality of Experience: First, SenseWear facilitates using more suitable sensors on another device when the smartphone's sensors are limited. For instance, smartphone games, such as car racing games, are often controlled by the user tilting the screen to different angles. However, viewing the screen at various angles and hand positions can be taxing on users. By integrating wearables, the user could control the game in more ergonomic ways, such as using their hands with the phone on a stand, thus keeping the phone steady and in a more comfortable position.

Improving User-Interaction Modalities: Existing mobile apps have pre-defined user interaction modes based on how the developer writes the app. Users' needs vary, and the original interaction mode may not be well-suited for every user. SenseWear facilitates using more suitable sensors on another device when the smartphone's sensors are limited. For example, a user may control the smartphone, such as locking the screen, adjusting volume, by waving gestures with the smartwatch. It can be more convenient especially when the smartphone is less reachable than the smartwatch. In this case, developers may utilize ambient light sensors on the smartwatch to control the behaviors on the smartphone. However, customizing the input of an application for all possible models of interaction is infeasible and requires considerable developer effort. SenseWear defines an easy-to-write metaprogram wherein a developer, or even a savvy user, can customize user interactions without app source code.

3 SENSEWEAR DESIGN

In this section, we first introduce the architecture of SenseWear and each component in detail. We then provide a high-level overview of the design of SenseWear, while explaining how SenseWear performs sensor virtualization and the steps needed by a user to apply SenseWear to an existing app.

3.1 Overall Architecture of SenseWear

Figure 1 shows the overall architecture of SenseWear. At its core, SenseWear aims to virtualize sensing so that a sensor from a remote device can replace local sensors to overcome hardware limits, change interaction patterns, and improve application QoE.

To this end, SenseWear is a sensor-sharing platform to virtualize wearable sensors, according to the policies specified by a developer. SenseWear enables wearable sensors to be utilized by any smartphone apps without modifying existing application code. The developer only needs to specify these policies using a simple metaprogram that requires a simple effort for sensor sharing.

There are two challenges in realizing the architecture: (i) mapping sensors across heterogeneous devices to meet different scenarios, e.g., mapping between the same types or different types of sensors, (ii) designing an easy-to-write metaprogram language that reduces the burden on the developer. We first describe the architecture and then discuss the SenseWear design that solves these challenges. The components of SenseWear are highlighted in Figure 1 and we explain the role of each below.

Metaprogram: A metaprogram is an easy-to-write program for any user or developer to customize their preferred way of using

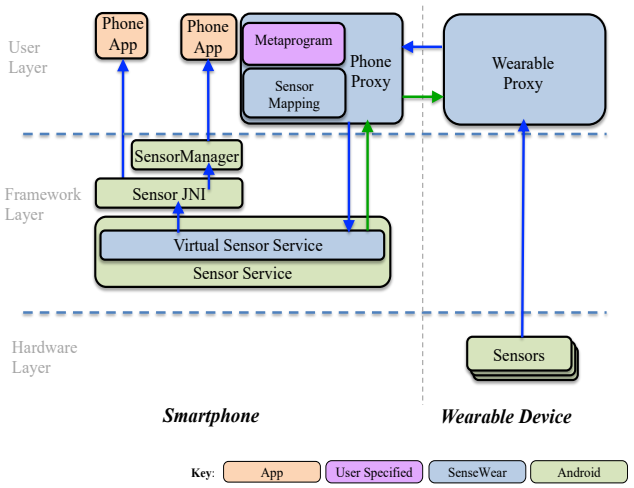

Figure 1: Overall architecture of SenseWear. The blue and green lines illustrate the data flow across devices. The blue lines indicate how smartphone apps utilize wearable sensors as input, such as accelerometers; the green lines show how SenseWear regards the wearable as an output, such as the vibrator.

wearable sensors to interact with phone apps. It consists of XML code, and optional Java code, which depends on which scenarios the users want to use the wearable sensors.

Phone Proxy: Phone Proxy is a long-running background service on the smartphone. It is the main service in SenseWear that is in charge of all the necessary jobs as described in Section 3.2, including loading and parsing of metaprogram, communication with Wearable Proxy, maps sensor data for phone use, etc.

Sensor Mapping: Sensor Mapping is a sub-component in Phone Proxy that defines how sensors are mapped between heterogenous devices. It calibrates or normalizes sensor data obtained from one device to the other device.

Wearable Proxy: The Wearable Proxy acts as a watch app that runs on foreground to provide a simple User Interface and gives necessary directives to users when necessary. For example, it asks a user to maintain the wearable device in a stable position for initial calibration. As discussed earlier, Phone Proxy receives sensor data from Wearable Proxy, based on the metaprogram specification on the Phone Proxy.

Virtual Sensor Service: To enable existing apps to seamlessly use remote sensor data, we need to intercept and replace the local sensors with remote sensor data. One option is to intercept the sensor API call at a relatively higher layer such as the Framework API layer [32]. However, many applications, especially gaming applications, use sensors via Jave Native Interface (JNI) [7] rather than Framework API. JNI circumvents Java API and JVM to deliver efficient instructions to the operating system. The mobile operating system is designed in a way that compels all sensors to go through the sensor service for dispatching sensor data. Therefore, we add Virtual Sensor Service at the Framework Layer as the sensor interception point.

3.2 SenseWear Workflow

Figure 2 shows the actions that a user needs to follow to apply SenseWear to an existing phone app.

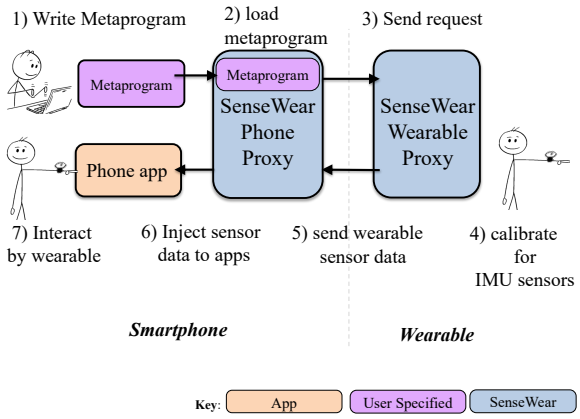

Figure 2: The overview of how SenseWear performs the sensor virtualization and what a user needs to do, in order to apply SenseWear, from a high-level perspective.

User preparing phase: (1) A user starts by writing a *metaprogram* to specify which local sensor(s) needs to be substituted by wearable sensor(s) and specifies how the wearable sensor(s) would be used on the phone. The user then uploads the metaprogram to a specified directory on the smartphone, via Android *adb* [1] command.

SenseWear parsing phase: (2) SenseWear Phone Proxy, which is a daemon process of SenseWear running on the phone, loads the metaprogram from storage and parses the metaprogram. (3) SenseWear Phone Proxy sends a request to Wearable Proxy to request sensor information as specified in the metaprogram.

User calibration phase: (4) If SenseWear Phone Proxy asks for IMU sensors, Wearable Proxy will prompt the user to maintain the wearable device in a static position for 30 seconds for calibration. This calibration process calculates the difference between the orientation of the phone and wearable devices and converts the wearable IMU sensor data into a form compatible with the phone. (5) The Wearable Proxy sends sensor data back to the Phone Proxy.

SenseWear runtime phase: (6) After receiving the sensor data, the Phone Proxy finishes the calibration process, and keeps converting incoming sensor data. Finally, SenseWear injects the ultimate sensor data to the operating system, which redirects the sensor to the smartphone app.(7) The user interacts with the original phone app via the wearable device, without modifying the phone app.

We note that users need to spend a limited amount of effort on sensor customization. The user's involvement is limited to choosing the interaction type in the metaprogram and if necessary to take actions as requested by the Wearable Proxy during the calibration phase for bootstrap.

4 METAPROGRAM

One of the goals of SenseWear is to help developers utilize wearable sensors in their applications with minimal engineering effort. Developers can specify minimal required information in a metaprogram, such as semantics mapping and sampling rates. SenseWear parses the metaprogram and performs the other steps involved in utilizing it, including mapping and transmitting sensor data.

SenseWear abstracts the set of sensor information that is required to virtualize sensing and specifies semantic mapping when needed.

```
<metaprogram>
<appname>waveLock</appname>
<sensor>
<sensor_type_from>ambient_light</sensor_type_from>
<sensor_type_to>proximity</sensor_type_to>
<override>onSensorOverride</override>
<freq>UI</freq>
</sensor>
```

(a)

```
public double[] onSensorOverride(double[] watchData) {
    /* Convert ambient light sensor data to
     *  proximity sensor
     * proximity sensor has only range of [0,5]
    **/
    if (watchData[0] > 5) watchData[0] = 5;
        else watchData[0] = 0;
        return watchData;
    }
}
```

(b)

Figure 3: Code (a) and (b) is a metaprogram that replaces a phone proximity sensor with a smartwatch ambient light sensor. The XML tag *Override*, specifies a Java function name, such as, onSensorOverride, that overrides for users/developers preferable sensor translation. The code snippet (b) shows the overrided function orchestrates with code (a) that converts watch sensor data, i.e., ambient light sensor, into the phone proximity sensor data.

The code snippet attached below shows an example of a metaprogram in SenseWear. A key feature of SenseWear is that each sensor is classified into a particular sensor type, such as accelerator, proximity sensor, or heartbeat rate sensor. By specifying the type of sensors and their corresponding mapping rules to fit into the local device, developers can easily adapt existing apps to utilize wearable sensors seamlessly and transparently. In some cases, the developer may have to write a function to specify the mapping. We let the developers specify this mapping in Java, as shown in Figure 3(b).

sensor_type_from and **sensor_type_to**: This group of attributes specifies the type of sensors the system is getting from and mapping to. But if both *sensor_type_from* and *sensor_type_to* are used, then the system replaces local *sensor_type_to* sensor with wearable *sensor_type_from* sensor. The example metaprogram shown replaces smartphone proximity sensor data with wearable ambient light sensor data.

Java programming for deep customization: To deep customize the sensor mapping, SenseWear enables developers to reserve the original ways of interacting with different sensors. As the code snippet (a), (b) show, the developer wants to use a wearable ambient light sensor to replace the smartphone proximity sensor. To achieve this goal, SenseWear provides a new callback API *onSensorOverride* in code (b) where developers can specify their preferred way of customizing the mapping between two sensors.

5 SENSOR MAPPING

We apply different approaches depending on the two possible categories of sensor mapping – replacing sensors of the same kind and that of different kinds.

5.1 Mapping sensors of the same kind

One-to-one mapping applies when a user wants to replace a local sensor on the smartphone with the same type of sensor from a wearable devices. For example, a user may prefer to have his/her smartwatch to enable the vibration mode when a phone call comes in, rather than to enable the smartphone, as wearable devices are more attached to body and can thus give more timely prompts. In this case, the user could just utilize the wearable sensor data, such as vibrators, as vibrators across devices share the same pattern of vibration. [4]. A user may also wish to replace the smartphone accelerometer sensor with a wearable accelerometer. Figure 4 shows how different orientation angles can produce different values for sensors like gyroscopes, accelerometers, magnetometers. Therefore, directly using the wearable sensors on the smartphone would cause the phone apps to behave incorrectly.

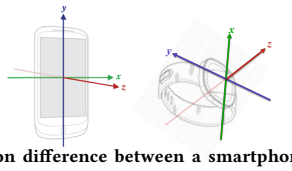

Figure 4: Orientation difference between a smartphone (on the left) and a smartwatch (on the right). We can see that these two devices have different initial orientations, which will make the smartwatch motion sensor data not work correctly to the smartphone.

We represent the transformation of sensors to that of another device as a type of affine transform [15]. We note that users tend to intuitively use movements of similar lengths, rotations and lateral movement for the same type of action on the same device. Across devices, we argue that there is a one-to-one mapping of such movement for each type of gesture. An affine transform can effectively "map" such movement of one type conveniently into another. Formally, let X_w be the output given by the sensors of a wearable device, and X_p be the output given by that of a smartphone. Then, using an affine transform matrix A, we can represent this as a matrix product, i.e., $X_p = AX_w$. Now, we need to identify the right values of the affine transform matrix A. Doing this directly is challenging, because of the following two reasons. First, the sensors have some amount of oscillating values. Second, the sensors also have different sampling rates. Accurate mapping requires to solve these challenges.

We resolve this challenge in the following way. We observe empirically that the relationship between the gyroscope and accelerometer values of the smartwatch and the smartphone is linear. We first use linear interpolation to ensure that the number of samples is equal. We then use a linear-regression [13] based learning of the affine transform matrix. We use linear regression because the affine transform matrix provides a linear mapping of the relationship between the sensors across devices. However, the exact mapping among the sensors may differ depending on the type or direction of movement. Thus, we use linear regression, with different parameters used for each type of movement. For each distinct gesture, we utilize linear regression using least-squares method to get a particular affine transform matrix.

SenseWear calculates the affine transform matrix using the initial data collected during bootstrap or calibration. During bootstrap, the UI on SenseWear first prompts the user to remain still with the smartphone and wearable devices for 30 seconds, to allow SenseWear to collect IMU sensors and calibrate based on the discussed approach.

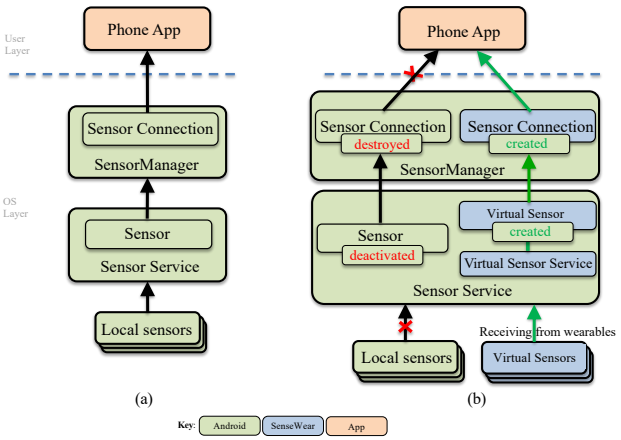

Figure 5: (a) The original design for sensor dispatching mechanism in Android system. (b) SenseWear design for sensor dispatching, highlighted on the right half. The left half shows that sensor dispatch connection would be destroyed by de-registering a sensor.

5.2 Mapping sensors of different kinds

This case occurs when a developer wants to replace the local sensor with a different type of wearable sensor. For example, some smartphone applications like Proximity Service [9] require a proximity sensor, which detects the distance between the object and the phone screen. Those apps act as a gesture controller so that users hovering on top of the screen can trigger the phone to make a phone call or just pick up a phone. To make full use of wearable sensors, users can hover on their smartwatch screens to trigger such phone actions. However, smartwatches only have ambient light sensors, which act similar to proximity sensors. But an ambient light sensor detects the light luminance rather than distance. However, proximity sensors and ambient light sensors yield similar data, as distance and luminance are strongly correlated. To support this use-case, SenseWear can simply map the ambient light sensor on the watch to the proximity sensor on the phone. This mapping rule is specified as a mapping rule in the metaprogram, to tell SenseWear to keep converting the wearable ambient light sensor data to proximity sensor data. The mapping rule specifies how to convert the values from one sensor type to another.

6 IMPLEMENTATION

We implemented SenseWear on Android OS 7.1.2 and Android Wear 2.0 smartwatch OS. We evaluate our prototype using a Nexus 5 phone and Sony Smartwatch 3. Other mobile operating systems such as iOS, have a similar architecture for sensor framework design, and it is possible to replace the Android watch with Apple watch by installing watch proxy.

Table 2 lists those sensors supported by SenseWear. While SenseWear can support a wider range of sensors than the listed ones, the implementation of SenseWear was limited by the hardware. For instance, Sony Smartwatch 3 does not sample heart rate data. Note that supporting more sensors in SenseWear can be done by additional software engineering, similar to the way we have mapped the sensors.

The communication between Phone Proxy and Wearable Proxy is based on Google Message Service(GMS) [6], which is a commonly used library to connect Android phones and watches via Bluetooth. Based on Rio [12], batching network requests is a common approach to reduce the latency of cross-device communication. Therefore, the phone proxy and watch proxy are sending the sensor data in batches with three sets of sensor data in a batch.

Sensor	Smart-phone	Smart-watch	Sensor	Smart-phone	Smart-watch
Accelerometer	✓	✓	Gyroscope	✓	✓
Magnetometer	✓	✓	Humidity	✓	✓
Vibration	✓	✓	Proximity	✓	
Screen Touch	✓		Ambient Light		✓

Table 1: Sensors supported by SenseWear.

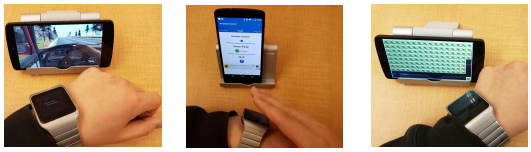

Figure 6: This figure shows four use cases that we implemented with SenseWear (a) replace the phone accelerometer with a smartwatch accelerometer while playing the game, RacingInCar; (b) control the smartphone using a smartwatch ambient light sensor replacing the phone proximity sensor, for AirGestureControl app; (c) using the smartwatch vibrator to get haptic feedback in place of smartphone, with HapticTest app.

App	LoC	Mapping Type
RacingInCar	10	Use wristwatch to control car in game
RacingFeverMotor	10	Use wristwatch to control car in game
AirGestureControl	18	Use wristwatch to control smartphone
WaveLock	18	Use wristwatch to lock and unlock smartphone
HapticTest	10	Vibration from phone to smartwatch
VibrationTest	10	Vibration from phone to smartwatch

Table 2: This table shows the apps used, along with the LOC for their metaprograms and the type of mapping we created to enable SenseWear for adapting

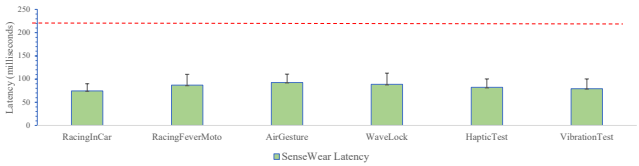

Figure 7: This figure shows the duration it takes from the sensor generated on a smartwatch to the sensor received on a smartphone. The red dashed line indicates the threshold of 220ms.

Virtual Sensor Service runs as a system-level thread, just like SensorService [2]. Virtual Sensor Service keeps listening to requests from Phone Proxy by *epoll* [5]. We also updated *SystemSensorManager* [3] in Android to expose a new API, *registerVirtualSensor* for developers to use. Virtual Sensor Service is able to cover all traditional sensors, such as IMU sensors, proximity sensors.

6.1 Use Case Implementation

We utilized six different types of apps to show how SenseWear can enhance existing smartphone apps by sharing remote sensors. Table 2 shows the lines-of-code required for those six apps utilizing wearable sensors. We could conclude that SenseWear is able to reduce the repetitive toil work for sensor developing, so that minimizes the development.

6.2 Sensor latency for apps

To measure the latency that an application incurs by using remote sensors, we measure the average time taken for the app to receive sensor data from the remote sensor. To minimize the time drift between two devices, we established a local NTP [8] server on a LAN network. This environment enables the devices to have a sub 10-millisecond error drift.

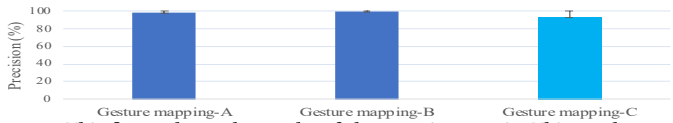

Figure 8: This figure shows the results of the mapping matrix. This result shows that mapping/calibration matrix works well with over 92% accuracy.

Figure 7 shows that the latency incurred for remotely receiving sensors is about 80 ms for the first six apps and under 200 ms for the last two. According to the psychophysics study [21] on human-perceived response time, users cannot perceive latencies under 220ms. That means even though replacing local sensor data using remote data does incur delays, this delay is acceptable.

6.3 Sensor Mapping performance

We test the working of one-to-one same-kind sensor mapping approach works by measuring the accuracy of mapping the phone and watch orientations. We measure the accuracy along each of the three possible axes, as well as the euclidean error in percentage. Let p_x and p'_x be the actual and predicted value of one sensor data point along axis x. Then, the error e_x is given by:

$$e_x = (|p'_x - p_x|)/p_x \times 100.$$

The root mean squared error e is given by: $e = \sqrt{e_x^2 + e_y^2 + e_z^2}$. We report the values of e_x, e_y, e_z and e for three gestures.

Figure 8 shows the errors for mapping of three different movements. Gesture mapping A represents the rotation of wrists anticlockwise and moving the phone left. Gesture mapping B represents the raising of arm and moving the phone forward. Gesture mapping C maps the movement of both smartwatch and phone both horizontally to the right.

Our experiments show that this technique can translate the sensors with a calibration error rate of at most 10% that is considered acceptable [22]. We note that the euclidean error in each case is at most 10%. We also observe that the error is relatively higher (around 10%) along a single axis while being significantly lower along the other axis. We explain this by observing that the user movement is usually along a particular axis, so the values along one axis remain relatively stable. Note that while a more complex model than a linear model would increase the accuracy, it would also make it necessary to get additional training data, which would hurt usability. We have also anecdotally seen that users report a significantly more positive experience when interacting with the smartwatch using this calibration.

7 RELATED WORK

Remote access support Many solutions focus on remote access to a server or desktop, like RDP [24] and VNC [26], albeit without processing sensor data. Fluid [28] and UIWear [35] focus on virtualizing and sharing UI across devices, which also reduces programming effort. SenseWear complements their functionality by facilitating sensor programming, thus making remote devices easier. Rio [12], Mobile Plus [29] and M2 [11] also enable resource sharing across devices. Specifically, Rio provides a general and efficient way to virtualize all I/O peripherals, without considering the compatibility issue across devices. Mobile Plus shares resources at the functionality layer instead of raw data. M2 shares I/O data, such as display, audio, sensors, across heterogeneous devices to adapt to several use cases, but it does not present a general solution to address the problem of heterogeneous data.

Cross-device interaction: To enable richer interaction, multiple studies have proposed utilizing multiple devices to improve cross-device interactions. For example, Highlight [27] and Panelrama [36] dispatch parts of UI to different form factors. WatchConnect [18], TouchSense [19] and Serendipity [34] define a new series of gestures to operate handheld devices with smartwatches. UIVoice [33] enables smart home voice agents to act as remote agents to control or interact with mobile apps seamlessly. Harmonious Haptics [20] acts as additional tactile displays for smartphones. Duet [14] presents a novel way of smartphone interaction using smartwatch motion sensors. Unlike SenseWear, these studies do not focus on helping the developers choose the sensor type of sensor for interaction.

Interaction and Gesture Mapping: Gluey [31] and MultiFi [16] design new interaction techniques specifically for Head Mounted Display (HWD) to operate other form factors. However, these techniques have not considered using the HWD sensor data for existing handheld apps. Hamilton et al. [17] present a way for users to migrate UI between devices without migrating or reusing sensor data. Yun et al. [37] demonstrate to use the smartphone as a mouse to the larger form factor such as Smart TV. However, these techniques only utilize sensors to recognize or analyze gestures and do not aim to adapt different gesture approaches to the same apps.

8 CONCLUSIONS AND FUTURE WORK

In recent years, a large number of sensing devices such as smartwatches have become commercially available. However, these wearable sensors are not fully utilized by most existing applications. We circumvent this limitation by designing SenseWear, which allows existing apps to utilize sensors on wearable apps seamlessly. SenseWear is a sensor sharing platform that allows a developer to easily specify how wearable sensors can be used to extend smartphone applications. We evaluate SenseWear by creating six SenseWear-enabled apps and conducting latency evaluations, and count the lines-of-code for enabling wearable sensors, respectively. Our evaluations show that SenseWear significantly improves the functionality of applications, with minimal effort from the developer, and without significant latency costs.

As future work, we plan to extend our work to more sensors. Currently, SenseWear only maps smartwatch sensors. We will extend SenseWear to seamlessly integrate with more devices, such as earpods and fitness trackers. Intuitively, such mapping is likely to require more advanced techniques based on machine learning. Specifically, a few simple gestures can be recognized by smartphones using techniques like incremental decision trees [30] or CNN [23]. However, calibrating sensors with different forms of signals while still retaining its usability is a major challenge. We also plan to optimize power consumption by disabling the local sensors being replaced by wearable sensors. This is non-trivial, because disabling local sensors could affect or even stop the apps function from functioning correctly. Thus, SenseWear needs a new technique of disabling such sensors.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback on the work. This work was supported in part by NSF grants CNS-1717973 and CNS-1718491.

REFERENCES

[1] [n. d.]. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb .

[2] [n. d.]. Android Sensor Stack. https://source.android.com/devices/sensors/sensor-stack.

[3] [n. d.]. Android SensorManager. https://developer.android.com/reference/android/hardware/SensorManager.

[4] [n. d.]. Android Vibrator API Reference. developer.android.com/reference/kotlin/android/os/Vibrator.

[5] [n. d.]. Epoll Wikipedia. https://en.wikipedia.org/wiki/Epoll.

[6] [n. d.]. Google Messaging Service. https://developers.google.com/android/reference/com/google/android/gms/gcm/package-summary.

[7] [n. d.]. Java Native Interface. https://developer.android.com/training/articles/perf-jni.

[8] [n. d.]. ntpd. http://doc.ntp.org/4.1.0/ntpd.htm.

[9] [n. d.]. Proximity Service. https://play.google.com/store/apps/details?id=ss.proximityservice&hl=en_US&gl=US.

[10] [n. d.]. Two in three phones to come with NFC in 2018. https://www.nfcworld.com/2014/02/12/327790/two-three-phones-come-nfc-2018/.

[11] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh. 2019. Heterogeneous Multi-Mobile Computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. Association for Computing Machinery, New York, NY, USA, 494–507. https://doi.org/10.1145/3307334.3326096

[12] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/2594368.2594370

[13] Fred L Bookstein. 1975. On a form of piecewise linear regression. *The American Statistician* 29, 3 (1975), 116–117.

[14] Xiang 'Anthony' Chen, Tovi Grossman, Daniel J. Wigdor, and George Fitzmaurice. 2014. Duet: Exploring Joint Interactions on a Smart Phone and a Smart Watch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/2556288.2556955

[15] Rafael C. Gonzalez and Richard E. Woods. 2006. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[16] Jens Grubert, Matthias Heinisch, Aaron Quigley, and Dieter Schmalstieg. 2015. Multifi: Multi fidelity interaction with displays on and around the body. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 3933–3942.

[17] Peter Hamilton and Daniel J. Wigdor. 2014. Conductor: Enabling and Understanding Cross-device Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. ACM, New York, NY, USA, 2773–2782. https://doi.org/10.1145/2556288.2557170

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

[19] Da-Yuan Huang, Ming-Chang Tsai, Ying-Chao Tung, Min-Lun Tsai, Yen-Ting Yeh, Liwei Chan, Yi-Ping Hung, and Mike Y. Chen. 2014. TouchSense: Expanding Touchscreen Input Vocabulary Using Different Areas of Users' Finger Pads. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. ACM, New York, NY, USA, 189–192. https://doi.org/10.1145/2556288.2557258

[20] Sungjae Hwang, John Song, and Junghyeon Gim. 2015. Harmonious Haptics: Enhanced Tactile Feedback Using a Mobile and a Wearable Device. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 295–298.

[21] Donald R.J. Laming. 1968. Information theory of choice-reation times. https://doi.org/10.1002/bs.3830140408

[22] Huy Viet Le, Sven Mayer, Patrick Bader, and Niels Henze. 2017. A smartphone prototype for touch interaction on the whole device surface. In *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*. ACM, 100.

[23] Zhi Lu, Shiyin Qin, Lianwei Li, Dinghao Zhang, Kuanhong Xu, and Zhongying Hu. 2019. One-Shot Learning Hand Gesture Recognition Based on Lightweight 3D Convolutional Neural Networks for Portable Applications on Mobile Systems. *IEEE Access* 7 (2019), 131732–131748. https://doi.org/10.1109/ACCESS.2019.2940997

[24] Todd R Manion, Ryan Y Kim, and Kestutis Patiejunas. 2014. Remote desktop access. US Patent 8,776,188.

[25] Greg Milette and Adam Stroud. 2012. *Professional Android Sensor Programming* (1st ed.). Wrox Press Ltd., GBR.

[26] Kazuhiro Nakao and Yukikazu Nakamoto. 2012. Toward remote service invocation in android. In *2012 9th international conference on ubiquitous intelligence and computing and 9th international conference on autonomic and trusted computing*. IEEE, 612–617.

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

[28] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R. Jeong, Steven Y. Ko, and Insik Shin. 2019. FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 42, 16 pages. https://doi.org/10.1145/3300061.3345443

[29] Sangeun Oh, Hyuck Yoo, Dae R Jeong, Duc Hoang Bui, and Insik Shin. 2017. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 332–344.

[30] Surbhi Saraswat, Ashish Gupta, Hari Prabhat Gupta, and Tanima Dutta. 2020. An Incremental Learning Based Gesture Recognition System for Consumer Devices Using Edge-Fog Computing. *IEEE Transactions on Consumer Electronics* 66, 1 (2020), 51–60. https://doi.org/10.1109/TCE.2019.2961066

[31] Marcos Serrano, Barrett Ens, Xing-Dong Yang, and Pourang Irani. 2015. Gluey: Developing a head-worn display interface to unify the interaction experience in distributed display environments. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*. ACM, 161–171.

[32] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2750858.2804260

[33] Ahmad Bisher Tarakji, Jian Xu, Juan A. Colmenares, and Iqbal Mohomed. 2018. Voice Enabling Mobile Applications with UIVoice. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking* (Munich, Germany) *(EdgeSys'18)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/3213344.3213353

[34] Hongyi Wen, Julian Ramos Rojas, and Anind K. Dey. 2016. Serendipity: Finger Gesture Recognition Using an Off-the-Shelf Smartwatch. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. ACM, New York, NY, USA, 3847–3851. https://doi.org/10.1145/2858036.2858466

[35] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E. Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. ACM, New York, NY, USA, 369–382. https://doi.org/10.1145/3117811.3117819

[36] Jishuo Yang and Daniel Wigdor. 2014. Panelrama: Enabling Easy Specification of Cross-device Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. ACM, New York, NY, USA, 2783–2792. https://doi.org/10.1145/2556288.2557199

[37] Sangki Yun, Yi-Chao Chen, and Lili Qiu. 2015. Turning a Mobile Device into a Mouse in the Air. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (Florence, Italy) *(MobiSys '15)*. ACM, New York, NY, USA, 15–29. https://doi.org/10.1145/2742647.2742662