AFV: Enabling Application Function Virtualization and Scheduling in Wearable Networks

Harini Kolamunna,[1] **Yining Hu, Diego Perino,**[2,4*] **Kanchana Thilakarathna,**[1] **Dwight Makaroff,**[3,5*]
Xinlong Guan and Aruna Seneviratne[1]
Data61/CSIRO-Australia
[1]University of New South Wales, [2]Telefonica Research, [3]University of Saskatchewan
Email: (firstname.lastname)@data61.csiro.au, [4]diego.perino@telefonica.com, [5]makaroff@cs.usask.ca

ABSTRACT
Smart wearable devices are widely available today and changing the way mobile applications are being developed. Applications can dynamically leverage the capabilities of wearable devices worn by the user for optimal resource usage and information accuracy, depending on the user/device context and application requirements. However, application developers are not yet taking advantage of these cross-device capabilities.

We thus design AFV (Application Function Virtualization), a framework enabling automated dynamic function virtualization/scheduling across devices, simplifying context-aware application development. AFV provides a simple set of APIs hiding complex framework tasks and continuously monitors context/application requirements, to enable the dynamic invocation of functions across devices. We show the feasibility of our design by implementing AFV on Android, and the benefits for the user in terms of resource efficiency and quality of experience with relevant use cases.

ACM Classification Keywords
C.5.3 Microcomputers: portable devices; C.2.4 Distributed Systems: Distributed applications

Author Keywords
adaptation; smart wearable devices; energy utilization; context monitoring; middleware frameworks

INTRODUCTION
Numerous wearable devices such as smartphones, tablets, smartwatches, and fitness bands form Personal Area Networks (PANs) as illustrated in Figure 1. These devices contain a rich set of sensors that may continuously monitor personal attributes and upload them to Internet servers to provide value-added services. Some devices may each contain sensors that enable the same functionality. For instance, a fitness band, a smartwatch and a smartphone are each likely to have an accelerometer, a gyroscope, and a heart rate monitor. Similarly, each device may have direct Internet connectivity, providing multiple Internet access points. Furthermore, some wearable devices (Tier 1 devices in Figure 1), will have sufficient computing resources to perform other functions such as data encoding, compression and encryption.

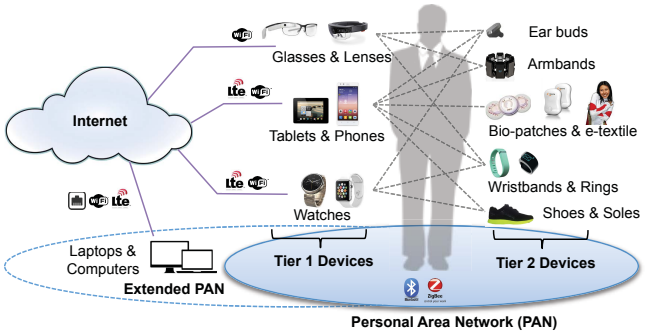


Figure 1. An overview of an personal wearable network.

The majority of popular wearable apps do not efficiently utilize the functionalities available on the devices. For example, the smartwatch pedometer will still use its own accelerometer when the battery level is low, despite an accelerometer being available on a fully-charged smartphone. Application developers rely on the APIs provided by the target device.[1]

To harness the collective capabilities of PAN devices, developers have to implement each app individually, incorporating device resources, the cost of running these functions in each device and communication costs. This necessarily increases the complexity of the mobile application development. We analyzed several popular wearable health and fitness tracking applications to identify the main challenges in building context-aware wearable applications, and to understand how effectively they are solved by existing apps. Developers tend to ignore the potential of using other available devices, compromising device and network lifetime, as well as the context-dependent utility of information.

Personal mobile applications are evolving from being a single program running on a smartphone, to being distributed across

*This work was done while the author was at Data61.

[1]These APIs abstract a large spectrum of functions (e.g., sensing, communication) that are actually implemented by the operating systems or the third party libraries and executed in the device where the application is running.

UbiComp '16, September 12-16, 2016, Heidelberg, Germany

DOI: http://dx.doi.org/10.1145/2971648.2971727

a set of personal smart devices [4]. Developers will need to leverage PAN resources to provide the best utility for the user. Therefore, a framework that takes user and device context into account, a capability realizable with the recent advancements in wearable computing, to utilize all PAN resources is needed.

In this paper, we present such a framework that extends the concept of network function virtualization [13] to device functions, then show its feasibility and advantages to users and application developers. In particular, we make the following main contributions:

- We design AFV: a framework for wearable application function virtualization and development of adaptive context aware wearable applications. AFV provides a simple set of APIs hiding complex framework tasks while enabling continuous context monitoring and adaptive function allocation across devices without developer or user involvement.
- We propose a greedy heuristic *function allocation* algorithm across devices, considering device/user/application preferences and function implementation costs. We evaluate the effectiveness and accuracy of the proposed algorithm through simulations driven by real experimental cost values.
- We implement AFV on Android to show the feasibility of our design, which can be accessed as an external library by application developers.
- We demonstrate AFV's ability to adapt to context changes and the benefits for the user through emulated set of relevant use cases in terms of energy consumption, network access quality, and monetary cost of bandwidth usage.

The rest of this paper is organized as follows. We begin with motivation via an exploration of current approaches of application development for wearable devices. Then we present the architecture and the context-aware adaptive function allocation algorithm and simulation evaluation. The implementation details and experimental validation of the framework with respect to adaptation and optimization are given next. Finally, we overview related work and complementary systems, and provide conclusions with future work.

MOTIVATION

We consider two dimensions of user utility: functional requirements (precision, accuracy), and performance requirements (energy consumption, latency). User experiences show that one or both of these characteristics are lacking in current applications [14, 21]. Our initial investigation of popular fitness applications shows that a lack of adaptability limits utility regarding the application's intended purpose and could reduce network lifetime significantly. Dissatisfied users will hinder wide deployment.

To determine under what circumstances current wearable fitness apps adapt their behaviour, we use a PAN that consists of an LG Nexus 5 smartphone running Android 6.0, and an LG Watch Urbane running Android 5.1.1. We chose the five most popular health and fitness tracking apps available on both smartphones and wearables (see Table 1) as they monitor user activities, report statistics to the user, store data on a device and upload them to a webserver.

We collected a trace of the execution logs (using Android Device Monitor) for evaluation. The API call `onSensorChanged` indicated sensing activity. To check the establishment of connectivity between the smartphone and the wearables, we enabled `Bluetooth HCI snoop log` on the phone to capture Bluetooth HCI packets and displayed the traces using Logcat. Connectivity establishment frequency was determined by checking the `switching from SNIFF to ACTIVE` log messages. We used Wireshark to analyze Internet transmission and identify the communication frequency between the application and its corresponding server. Function allocation and storage options were also reported for each application.

Function allocation

We first consider sensing, data storage and connectivity establishment functions. Table 1 shows that 3 out of 5 applications examined required the user to configure sensor usage across the available PAN devices manually via the smartphone (master). The applications on the wearable devices acquire the configuration from the smartphone automatically.

Most applications require the user to select a single sensor or a single device to perform a sensing operation. MyFitnessCompanion allows the step count and the heart rate to be independently associated to a given sensor/device pair, while in the other applications, the user must select a single device for all measurements. Only UP allows multiple devices to take measurements in parallel.

Data upload and local data storage functions are fixed and pre-configured. All applications use the smartphone, the only exception being WearRun, which does not communicate.

Observations: *Most functions are automatically performed on the smartphone, except sensing functions which can, in some cases, be preconfigured by the user. The smartphone acts as a master device and wearables are considered simple peripheral sensors, although they already have and will increasingly have even more capabilities to perform more advanced functions.*

Context awareness

We investigated whether the allocated functions change their behaviour according to the device and user context. We considered 4 contextual parameters: 1) execution mode (foreground, background), 2) battery level (low $< 20\%$, high $> 20\%$), 3) walking pattern (slow, fast), and 4) location (outdoor, indoor).

Sensing frequency depends on the sensor, mostly determined by the Operating System (OS). Indeed, the developer can specify a given sensing frequency, but this is only considered to be a hint and it is usually ignored. The sensing frequency of UP increases with the walking speed for both smartphone and smartwatch. All applications using motion sensors adapt their behaviour to changes in walking speed; this is a feature defined by the OS. In contrast, Sleep on the smartphone only varies the sensing frequency when the application execution mode changes from foreground to background.

Data exchange frequency is application-dependent as shown in Table 1. When data is sensed on multiple devices in parallel (e.g. UP) for a particular function, only one value is selected discounting all the resources consumed by the other devices.

Application	Activity	Device	Primary Sensors	Function Allocation	Data Exchange [Hz]	Sensing [Hz]	Upload [Hz]	Storage
UP	Walking	Phone Watch	Motion 9-axis	Manual Automatic	1/60	1.5 1.5	1/60 N/A	Daily N/A
MyFitnessCompanion	Walking	Phone Watch	GPS Motion	Manual Automatic	1/3	1 1	1/60 N/A	On-demand N/A
WearRun	Walking	Phone Watch	N/A Step counter	N/A Automatic	N/A	N/A 1.4	N/A N/A	N/A Automatic
Sleep	Staying	Phone Watch	Accelerometer Accelerometer	Manual Automatic	1/20	1.1K 2.2K	0.1/60 N/A	On-demand N/A
Cardiograph	Sitting	Phone Watch	Camera HR sensor	Automatic Automatic	On-demand	28.6 -	1/60 N/A	On-demand N/A

Table 1. Summary of function allocation and context-awareness of popular health and fitness applications.

Some applications modify the data exchange frequency when the execution mode changes. For instance, UP only exchanges data (0.016Hz) when the smartphone application is in the foreground; Sleep reduces the frequency from 0.05Hz to 0.008Hz when changing from foreground to background. This is due to two main reasons: 1) data aggregation on the smartphone is not required or can be performed at a low frequency when the app is in the background, as data is not actually reported to the user, and 2) existing tools force the user to define how an application should behave for the different execution modes.

Finally, data upload to an Internet webserver and data storage are agnostic to all context parameters. Data storage is often performed on demand by all applications. Moreover, all application functions are agnostic to battery level and location parameters. Developers would have to provide ad-hoc monitoring and adaptation engines.

Observations: *Existing applications do not provide context monitoring or dynamic adaptation to context changes. They leverage limited context awareness tools provided by the OS.*

Using context for the optimization of computation, sensing and communication resources has been widely studied in the research literature (e.g., [8, 11]). However, determining the user context accurately and efficiently has been a difficult task, which may require several lines of code (LoC) with a significant impact on the development complexity. As a consequence, most current applications do not implement adaptive function placement. Instead, functions are statically pre-defined by the developer or user-selected.

ARCHITECTURE

Therefore, if all available PAN resources and the user context can be leveraged, it should be possible to improve the usability and the quality of experience of the users significantly. We believe that this can be done by providing an AFV interface, which takes care of the context-aware function placement for utilization of resources available on multiple PAN devices. The AFV framework has the following features:

- **Usability:** minimizes the complexity for application developers and configuration for the user;
- **Optimality:** carries out the function placement to maximize the usability and user quality of experience;
- **Adaptability:** dynamically configures the system at runtime according to changes in the context, applications and user needs.

Current PAN devices can be divided into two broad categories, which we refer to as Tier 1 and Tier 2, as depicted in Figure 1. Tier 1 devices, (e.g. smartglasses, smartwatches, smartphones), are relatively more resourceful than Tier 2 devices. Tier 2 devices, (e.g. smartshoes and bio-patches) simply carry out sensing functions. The distinguishing features of Tier 1 devices are the following: a) availability of heterogeneous long-range network *connectivity* (WiFi, cellular), and b) ability to *process* sensed information. Thus, Tier 1 devices may be "in-charge" of making decisions as well as be equipped with a rich-set of sensors, multiple connectivity interfaces, storage and computing power to perform wide set of functions (e.g., compression, encoding, rendering, intrusion detection, firewall filtering, encryption), many of which are generic and can be executed in multiple PAN devices.

Figure 2 provides an overview of the main modules of the AFV framework. Every application registers requests for one or more virtual functions via the function APIs. Application function registration requests are managed by the *Function Manager* module, while the *Function Execution* module manages function invocation on the device identified by the *Decision Engine*. The *Context Monitoring* module periodically monitors device/user context, providing inputs to the decision engine. The *Communication Manager* maintains efficient communication among the AFV-enabled devices (Tier 1 devices). Tier 2 devices are managed via the OS of the paired devices.

AFV APIs

AFV provides two main types of APIs: function APIs that are executed during regular operation and preference APIs that are executed at application start-up.

Function APIs

For *each supported function*, AFV provides a specific API to the developer. As such, the API augments the existing API for functions on single devices by incorporating the necessary components to realize framework tasks. As an example, Figure 3 contains the AFV sensing function API, along with the API provided by the OS[2] for the same function on a single device. The `onSensorChanged` and `unregisterListener` APIs do not require any change with respect to OS APIs. The `registerListener` API requires simple modifications from the original API. First, the `maxReportLatencyUs` input is not only used as in the original API, but it also defines the data exchange frequency between applications and devices. Second,

[2]Android in the example

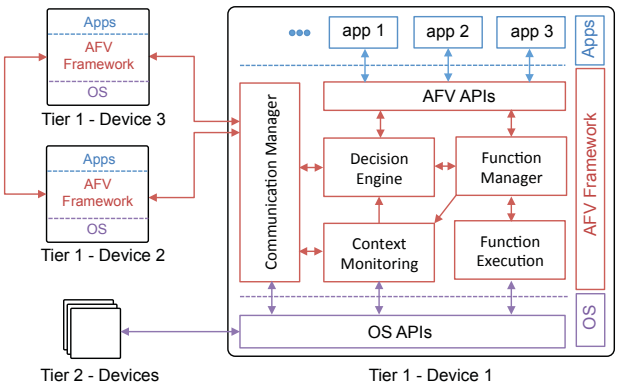


Figure 2. Overview of the AFV and logical connectivity among devices.

the `precision` input indicates the required measurement accuracy with respect to absolute correctness, provided as an list of contexts and ranges. The optional `mapping` parameter is a list of (context, device) pairs allows the developer to override the *Decision Engine* and select a particular device. We use a list of (name, value) pairs to represent a context.

Preference APIs

Each application allows the developer to specify valid context operating ranges for each virtual function and/or its implementations. In some cases, the user may wish to override the application's settings. Additionally, since the AFVframework covers the entire PAN, each device may have configurations set by the manufacturer or the user for every application running on the device. This is done using Preference APIs. The AFVprovides three types Preference APIs, namely application (`setAppPrefs()`), user (`setUserPrefs()`) and device (`setDevicePrefs()`).

The API is not directly exposed to the user, however, but is leveraged by the developer to take user preferences as inputs to the framework. For instance, the UI of the application could provide an interface with radio buttons or drop-down lists to allow the user to select forced or prevented device/context mappings. The user/device preferences are provided to each instance of the framework in the PAN via the *Communication Manager*. They are mostly used by the *Decision Engine* to identify valid functions and implementations for the current application context. These preferences are also used by the *Context Manager* to recognize context changes.

The configurations (application, user and device) could be conflicting. In order to enable a meaningful context for the optimization and mapping components, certain possible configurations may be mandated, or eliminated from contention. For example, iOS devices are only suggested to operate between 0 and 35 degrees C,[3] so any user specified contexts beyond this must be ignored. For the current version of the framework, we have selected the following order of priority: Device, User, and finally Application.

Function manager

The function manager is the main interface between AFV and the applications. For each virtual function $v \in V$, the function manager stores the set of devices D_v that provide an implementation of v. It also stores device function execution costs.

[3] https://support.apple.com/en-au/HT201678

Android

```
abstract void onSensorChanged(SensorEvent event)
boolean registerListener(SensorEventListener
                         listener, Sensor sensor,
                         int samplingPeriodUs, int
                         maxReportLatencyUs)
void unregisterListener(SensorEventListener listener,
                        Sensor sensor)
```

AFV

```
abstract void AFVonSensorChanged(AFVSensorEvent event)
boolean AFVregisterListener(AFVSensorEventListener
                            listener, AFVSensor sensor,
                            int samplingPeriodUs,
                            int maxReportLatencyUs,
                            List<Context, <int,int> >
                            precision, List<Context,
                            Device> mapping )

void AFVunregisterListener(AFVSensorEventListener
                           listener, AFVSensor sensor)

void setAppPrefs(AFVApplication applicationName,
                 AFVDevice deviceName, List<Context,
                <int, int> >)
```

Figure 3. Examples of APIs provided by AFV and by Android.

The cost is composed of two main factors; the cost of executing the function implementation f and the cost of exchanging inputs/outputs between the application and the implementation c. The cost usually represents relative energy consumption, but other types of costs are also allowed. A device must have a pre-defined list of supported functions and associated costs.

Each time a novel AFV-enabled active device joins the PAN, AFV first discovers the virtual functions supported by the device, and then it announces such functions and related costs to the entire network. Devices already in the network reply to the novel device with the list of virtual functions they support. Each active device announces virtual functions provided by passive devices connected to it, and sends periodic keep-alive messages with an updated list of functions provided.

Another key role of the *Function Manager* is mapping management and the data exchange between application registrations for virtual function v, $a \in R_v$ and the device selected to run the implementation $d \in D_v$. The mapping $R_v \mapsto D_v$ is updated each time the decision engine runs. The *Function Manager* aggregates the multiple requests for the same function by different applications and only invokes one function request for function execution. Data exchange (i.e., input/output) between applications and the framework is performed via a set of per-application function buffers to avoid contention among applications accessing the same virtual function. Inputs/outputs are then transferred to/from the device in charge of running the actual function by the communication manager. Any change in the list of registrations R_v, or available implementations or devices in D_v triggers the *Decision Engine*.

Function execution

This module is responsible for the execution of virtual functions in the devices selected by the *Decision Engine*. We assume that a single implementation for each function is available in each device and it is provided by the framework itself. However, AFV can be extended to enable per-device context aware function selection leveraging previously proposed

frameworks as CAreDroid [6], or allowing custom function implementations. Function input/output between applications and the device running the function is performed via the *Communication Manager* and the *Function Manager*. For functions producing continuous output via callbacks (e.g., sensing functions), output is reported to applications at the frequency specified by the developer. This achieved via aggregation and buffering mechanisms in the virtual function class.

Communication manager

This module is in charge of managing all AFV communications in the PAN. It centralizes communications among modules running on different devices, and can aggregate messages and batch data transfers to limit communication costs. As previously stated, communication among devices in the PAN is performed via Bluetooth or other similar low-powered wireless technologies.

Context monitoring

We consider the context monitoring component as an additional virtual function, which runs on a device capable of receiving information from sensors, either directly or indirectly, and communicating with the other devices. The context function operates as needed and only reports changes relevant to the *Decision Engine*. It is possible for context monitoring to be carried out on different devices, depending on the context induced from the sensor values.

Context monitoring is an essential element of AFV. Several implementations of context monitoring exist in the literature [2, 8]. These can be adapted for AFV. Due to space limitations, detailed design of context monitoring is not discussed further. The context monitor evaluates the cost of obtaining the measures and selecting the appropriate mechanism/sensor with which to obtain this information. It is also possible that the context may be obtained from the device's operating system, if those features at that level are enabled [24].

Since the framework is intended to be used across multiple devices and multiple applications, contexts are represented in the common format previously described as pairs of name and value. Context pairs can be defined per-application and per-device in the specific configuration files. The state of a particular context pair is expressed and stored as an enumerated type or string. The value field could be a) a threshold, b) ranges (e.g. moderate temperature could be 20-30 C), or c) a binary value (e.g. the device is either charging or discharging).

Decision engine

The *Decision Engine* determines the mapping of application function registration requests to actual function implementations across devices. First, the decision engine filters out infeasible function implementations determined by the current device context. For example, even if the GPS is available on a device, the decision engine will not map it with any location request if the current remaining battery capacity of the device is very low. Then, each function registration request is mapped to one of the feasible implementations such that the total cost of all PAN devices is minimal. The costs can be defined based on both app developer and user objectives. We categorize them as 1) monetary costs, 2) quality costs and 3) energy costs, which are further described in the next section.

The *Decision Engine* is triggered for a new assignment of functions by either 1) the *Context Monitoring* module on context change or context configuration update, and 2) *Function Manager* on function registration requests/implementation change. If it is triggered by the *Context Monitoring* module, the feasibility of the function implementations is revisited as well.

CONTEXT-AWARE FUNCTION ALLOCATION

We denote as $r_{a,v,d} \in R$ the registration for a virtual function $v \in V$, at a device $d \in D$, for an application $a \in A$, where A is the set of applications installed in the devices of the PAN. $R_v \subset R$ and $D_v \subset D$ represents the set of registrations and the list of devices providing implementations respectively for a given virtual function v. Thus, the objective of the *context-aware function allocation* is to map each function registration request to its implementations, i.e. $R_v \mapsto D_v$, such that it optimizes the total cost of executing the all requested functions.

Function costs

The function costs are related to the usability objective of the system, i.e. monetary, quality and/or energy, which can be defined either by the app developer or the user. For each objective, there are two types of costs associated with each function requests and its implementations; 1) communication costs and 2) implementation costs. If the objective is to optimize the monetary costs of the user, internal communication (e.g. Bluetooth), can be considered as zero. On the other hand, if the objective is to optimize the energy consumption of the devices, communication is not negligible. Usually, if the request is mapped to the device itself, the internal communication cost is zero. For the same function, the implementation cost can be different for multiple devices, e.g. the energy cost of activating the WiFi network interface compared to the total battery capacity on the smartphone is lower than on the smartwatch.

The implementation function costs v in a device d is represented as $f_{v,d} \in F_v$. Similarly, the communication costs between a given function registration request $r_{a,v,d}$ and an implementation on device d is denoted as $c_{r,d} \in C_r$.

Problem formulation

We first define a binary variable $m_{r,d}$ where $m_{r,d} = 1$, if function registration request $r_{a,v,d} \in R_v$ can be mapped to implementation on device $d \in D_v$, and $m_{r,d} = 0$ otherwise depending on the current context of the user and the device. For instance, even if the GPS sensor is implemented on the device $d_1 \in D_v$, it may not be able to map d_1 with any request if the current remaining battery capacity on d_1 is below the threshold. If no function implementation is available for a particular function registrations request, we remove that function from the problem formulation. That makes for all considered functions $\sum_{\forall d \in D_v} m_{r,d} = 1; \forall r \in R_v$. Given the set function registrations R_v and function implementations D_v and the associated costs $f_{v,d}$ as input, the optimal FUNCTION ALLOCATION PROBLEM (FAP) can be formulated as follows:

$$\text{Minimize} \left(\sum_{d \in D_v} y_d \cdot f_{v,d} + \sum_{r \in R_v} \sum_{d \in D_v} x_{r,d} \cdot c_{r,d} \right) \quad (1)$$

Subject to:

$$1.\quad \sum_{d\in D_v} x_{r,d} = 1; \qquad \forall r \in R_v$$
$$2.\quad m_{r,d} \geq x_{r,d}; \qquad \forall r \in R_v, \forall d \in D_v$$
$$3.\quad y_d \geq x_{r,d}; \qquad \forall r \in R_v, \forall d \in D_v$$
$$4.\quad y_d, x_{r,d} \in \{0,1\}; \qquad \forall r \in R_v, \forall d \in D_v$$

The sets of $x_{r,d} \in X$ and $y_d \in Y$ would be the solution of the FAP. $x_{r,d} = 1$ if the function registration request $r \in R_v$ is assigned to the device $d \in D_v$ and $y_d = 1$ if the device d is required to be activated to satisfy certain requests. Only mappable implementations will be assigned and each function registration request will be mapped to an implementation.

Solution to function allocation

When $m_{r,d}$ is given $\forall r \in R_v, \forall d \in D_v$, it is trivial to show that FAP is equivalent to UNCAPACITATED FACILITY LOCATION (UFL) problem where every function implementation D_v is a facility with $f_{v,d}$ facility opening cost and every function registration request R_v corresponds to a customer associated with $c_{r,d}$ service cost. This immediately follow that FAP is also an *NP-Hard* problem. However, there are number of approximation algorithms for the well-studied UFL problem. We build on the approximation algorithm proposed by Williamson and Shmoys [25] to take into account the use of valid mappings ($m_{r,d}$) after context aware constraints. The iterative greedy solution to FAP is described in Algorithm 1.

Algorithm 1 FAP(R_v, D_v, m, f, c)

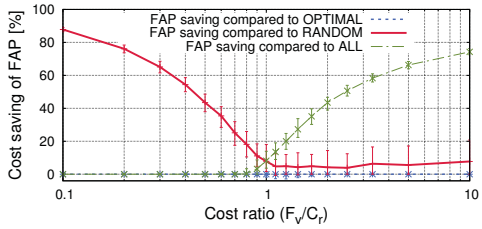
1. $S \leftarrow R_v$
2. $X \leftarrow \emptyset$
3. **while** $S \neq \emptyset$ **do**
4. Select $v \in D_v$ and $P \subseteq S$ s.t. $\forall p \in P : m_{r,d} = 1$ that minimize $\frac{f_{v,d} + \sum_{p\in P} c_{r,d}}{|P|}$
5. $S \leftarrow S - P$; $f_{v,d} = 0$
6. $(R_v \mapsto D_v) \leftarrow (R_v \mapsto D_v) + (P \mapsto v)$
7. **return** assignment $\sigma : R_v \mapsto D_v$

The algorithm iteratively selects a function implementation and the registrations associated to it for which there are valid mappings. Assigned registrations are then removed from the problem and the implementation cost set to 0. At each iteration, an implementation is selected as to minimize the total cost of function registrations that will be associated to the implementation. The algorithm can be efficiently realized by maintaining for each implementation the list of registrations not yet satisfied in increasing cost order; the set P minimizing the cost will be some prefix of this ordering [25].

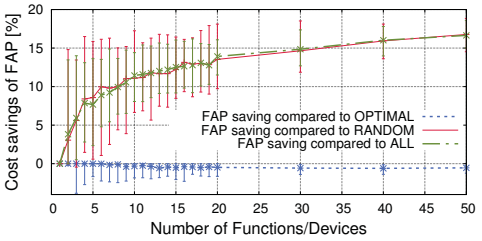
EVALUATION WITH SIMULATION

Evaluation methodology

We develop a custom simulator to analyze the effectiveness of FAP algorithm of our *Decision Engine*. We compare AFV function allocation against simple random device selection (RANDOM) and selecting all available devices (ALL), which are the two most common categories of today's wearable applications. The former represents applications that ask the user to select a device to execute a given function, e.g. MyFitnessCompanion app (Table 1). The latter represents applications that run the function on all devices in parallel, e.g. UP app. We also compare the approximation accuracy of FAP comparing the optimal (OPTIMAL) results obtained with the optimization problem solver Gurobi.[4]



(a) The impact of cost ratio F_v/C_r.



(b) The impact of no. of functions, $F_v/C_r = 1$.

Figure 4. Efficiency, accuracy and robustness of FAP algorithm.

We assume that function costs among the devices for the same function are normally distributed with standard deviation $\sigma = 0.1 \times \mu$ where μ is the average value. We change μ to obtain multiple cost values to evaluate the performance of FAP in various conditions.

Efficiency and robustness of FAP algorithm

We first report in Figure 4(a) the cost reduction obtained by our FAP algorithm with respect to RANDOM, ALL and OPTIMAL strategies as a function of the ratio of function implementation cost (F_v) to communication cost (C_r). Communication costs are incurred for any transmission of an individual sensor stream to the device executing the application.

We consider 5 active wearable devices in a PAN. Intuitively, if the communication cost is too high, it is more efficient to execute the function on each device, resulting in parallel apps with no coordination. This is reflected in the region where $F_v/C_r < 1$, i.e. ALL performs as good as OPTIMAL and FAP. However, FAP significantly reduces the cost compared to RANDOM selection. As F_v/C_r increases the significance of the communication cost decreases. Thus, executing a function in all devices becomes inefficient as there is potentially a device with a very low relative function implementation cost. Since there is a 1/5 chance of selecting the right device, RANDOM performs comparatively well with high standard deviation. FAP performs equally well (error is less than 1%) compared to OPTIMAL irrespective of the F_v/C_r value.

We now evaluate the impact of number of functions registration requests and devices in the PAN when the $F_v/C_r = 1$. Figure 4(b) shows that FAP increases its cost savings compared to both RANDOM and ALL along with the number of functions. On the other hand, FAP accuracy does not vary significantly

[4] https://www.gurobi.com

compared to OPTIMAL (error is about 2-3%). Overall, Figure 4 shows that the proposed FAP algorithm is often able to map the function registration requests to the optimal function implementation providing significant cost savings.

Prolong system uptime

In this subsection we analyze AFV effectiveness by considering network lifetime as example of quality metric. Indeed, prolonging system uptime is one of the biggest challenges with the current wearable devices. As lifetime of a device depends on its current SoC and current energy usage, we considered these 2 parameters for our simulations. We measured energy consumption of several application functions for sensing, communication,[5] and processing. We use a smartphone running Android 6.0 and LG Watch Urbane running Android 5.1.1 with 2300 mAh and 410 mAh battery capacities respectively, while the functions are reported in Table 2.

Function	Energy cost	
	Smartphone	Smartwatch
Sensing (NORMAL - UI - GAME - FASTEST) [mJ/s]		
Accelerometer	5.01 - 13.28 - 34.46 - 77.71	9.52 - 24.74 - 57.61 - 168.40
Gyroscope	11.71 - 20.33 - 36.44 - 80.15	16.23 - 33.34 - 60.44 - 181.90
Magnetometer	8.12 - 15.45 - 28.46 - 28.28	17.04 - 30.21 - 57.82 - 79.73
Connectivity (Per Byte [mJ/B] - High Power Idle [mJ] - Low Power Idle [mJ])		
Bluetooth	0.0095 - 305 - 300	0.0024 - 126.07 - 64.23
WiFi	0.00054 - 66 - N/A	0.00039 - 50 - N/A
Processing (Per Byte [mJ/B])		
Compression	0.01	0.0004
Encoding	0.00026	0.00025

Table 2. Energy cost associated to each function

The energy profile for each function is obtained with a Monsoon power monitor[6] directly connected to each device via USB. Energy usage is obtained by integrating the instantaneous power values which are calculated using current and voltage measurements from the USB interface sampled at a 0.2ms time interval. The energy usage of the experiment can be extracted by deducting the fixed energy offset of the background processes. Sensing energy is measured for multiple sampling frequencies that are offered by Android by default.[7]

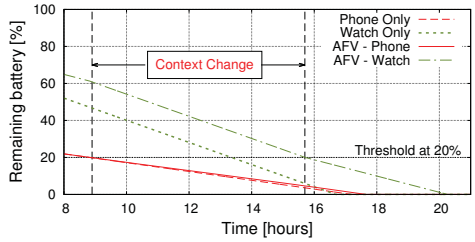
To simulate typical user behaviour, we assume the smartphone battery would completely drain in two days linearly and the smartwatch would last only one day. We consider the "sensing accelerometer" function and 60 second data synchronization frequency: the AFV *Decision engine* then minimizes the application energy consumption while respecting user preferences. Energy usage for any particular scenario that involves the above functions in Table 2 can be obtained by aggregating the energy values for each function. As an example for sense only on smartphone (Accelerometer NORMAL speed) and data synchronization frequency of one minute (13.5KB of data), we can get the energy consumption per minute from Table 2 as $f + c$=(5.01*60)+((0.0095*13500)+305+300)=1033mJ.

Figure 5 illustrates the battery drain profile for RANDOM selection, i.e. sense only on the smartphone or on the smartwatch,

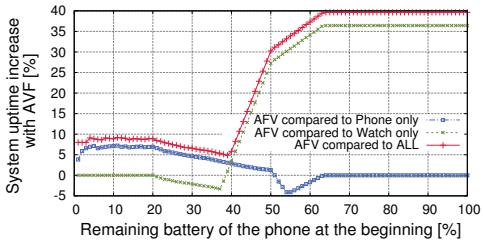
[5]Only WiFi *receive* measured, *transmit* is between 20 and 30% higher [26]

[6]https://www.msoon.com/LabEquipment/PowerMonitor/

[7]http://developer.android.com/reference/android/hardware/SensorManager.html



(a) Battery drain profiles



(b) Percentage increase in system uptime.

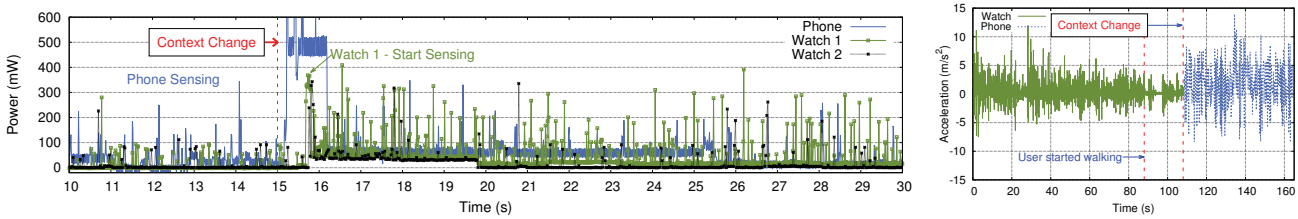
Figure 5. The impact of AFV on system uptime.

and when AFV is running on the system. Due to a lower relative impact on the smartphone, AFV selects the smartphone as the sensing device if the smartphone has sufficient SoC. However, if the smartphone's SoC drops below 20% (context change), the *Context Monitor* triggers the *Decision Engine* and sensing switches to the smartwatch if the smartwatch has sufficient SoC (Figure 5). To show this context change, we consider the following initial conditions: smartphone - 45% charge, smartwatch 100%. The smartwatch uptime increases by approximately 2 hours compared to only sensing on the smartwatch. The gain for the smartphone is about 1/2 hour compared to only sensing on the smartphone.

System uptime is defined as the time until at least one device drains out its battery. Since uptime gain is dependent on the initial SoC of devices, in Figure 5(b) we change the initial smartphone SoC. If the smartphone remaining SoC is greater than 60% at the beginning, AFV increases the system uptime about 35-40% compared to sensing on the smartwatch and on both devices. Due to sufficient battery capacity on the smartphone, AFV selects the smartphone most of the time. As a result, AFV does not increase the uptime compared to sensing only on the smartphone. AFV may marginally reduce the system uptime when the SoC of one or more devices drops below the threshold. Indeed, to minimize the application energy consumption while respecting user preferences, the *Decision Engine* selects the only available device (or the most energy efficient when both are under the threshold) although this solution may reduce system uptime. This can be observed when initial Phone SoC is around 35% and 55% in our scenario.

IMPLEMENTATION

There are two options for the implementation strategy for AFV: 1) a kernel module or 2) an SDK. The best performance would be achieved via integration of AFV into the kernel of the OS. In this manner, every application on any device could seamlessly take advantage of these facilities by flipping a bit and providing the correct configuration entries. The second option is a library in the form of an SDK and standalone



(a) Unplug phone from AC power. (b) User sitting vs walking.

Figure 6. Adaptation of AFV to context changes.

user-level application. This library would be compiled into the application and make IPC calls to the application when services are required. This strategy can be deployed without special access to the internals of the smartphone or the OS and allows developers to experiment with the features.

Our initial prototype follows the second option. For simplicity, all Tier 1 devices in the PAN run the same OS (e.g. Android); eventually, we envision a cross-platform implementation. We develop a library - AFVlib - that provides access to the AFV APIs to the developer once imported to application. To support AFV services multiple applications in parallel at the user-level, we opt to develop other components of AFV as a standalone application - AFVapp. At the time of installation of a AFV enabled application, it checks whether the AFVapp is already installed in the system; if not, it initiates the installation of the AFVapp. The prototype AFVlib and AFVapps consist of approximately 5000 lines of Java code.

A straightforward IPC mechanism (`BroadcastReceiver`) is used to communicate between AFV-enabled applications and our framework. The communication between AFV-enabled devices are implemented using `MessageAPI` as an Android `Service`. Context monitoring is also implemented as an Android `Service` which runs in the background and reports to the *Decision Engine* periodically or in change of context. Once the application registers a function (e.g. `AFVregisterListener(this, Sensor.TYPE_ACCELEROMETER)`), AFVlib sends a broadcast with an `Intent` about the function registration to AFVapp. The `BroadcastReceiver` in the AFVapp transfers the request to the *Function Manager*. The *Function Manager* then checks with the *Decision Engine* for the optimal placement of the function. The *Decision Engine* implements the FAP algorithm by leveraging efficient ordered data structures (i.e., `TreeMultimaps` and `ArrayLists`). *Function Execution* registers the function using Android APIs and returns data in `JSON` format to the *Function Manager*. The *Function Manager* aggregates data to one `JSON Array` per device and sends it via the *Communication Manager*. Each app that has a registered `listener` for the function will receive the data stream. In the remainder of the paper, we use AFVlib and AFVapp to evaluate the practical feasibility of AFV and user benefits with emulated use cases.

USE CASES WITH EXPERIMENTS

Experimental validation of AFV framework

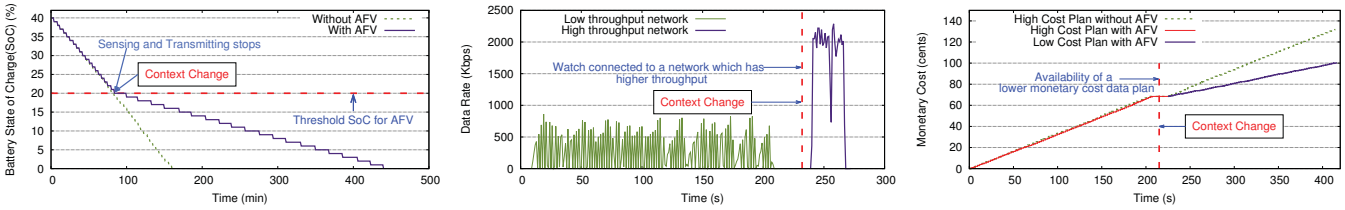
The AFVframework should have the ability not only to detect the context changes and select the optimal function allocation according to the context, but also to achieve that within the least possible time. Therefore, we first evaluate the FAP execution time on a LG Nexus smartphone and on a wearable (LG Watch Urbane). Our results show that the execution time is in the scale of tens of milliseconds. Especially for a realistic number of functions, e.g. 5 functions, it takes less than 1 ms to determine the optimal function allocation in both devices.

System adaptation to context changes. We experimentally evaluate AFV's dynamic adaptation to new configurations of functions allocation which is decided by FAP algorithm. For this evaluation, we created a PAN with three devices (one smartphone and two smartwatches). We consider the *sensing accelerometer* as the requested function and all three devices are subscribed to receive the sensed data. We emulate the change of context by plugging and unplugging the devices to AC power where the system objective is to optimize the energy consumption. Therefore, we measure the energy consumption of each device during the experiment using a Monsoon power monitor directly connected to each device via USB.

Figure 6(a) illustrates the power profile for three devices during the experiment. Initially, the smartphone is sensing at normal speed and sending data to the other two devices once a minute. Then, we emulated a context change (unplugging the smartphone from AC power) at $t = 15$ seconds. The smartphone broadcasts the context change to all devices in the PAN, which then triggers the *Decision Engine* on the smartphone to select watch-1 for sensing. The smartphone notifies the new function allocation to other devices and transfers the sensing function to watch-1. The smartphone stops sensing after watch-1 acknowledges it has taken over the sensing function.

The high power peaks of all devices after $t = 15$ seconds is due to the messages received and transmitted by each device, which is followed by high power idle states. The high power idle state is longer for the smartphone (until $t = 26$ seconds) compared to the watch (until $t = 20$ seconds). Figure 6(a) shows that the overhead of system adaptation to context changes is less than one second as the watch-1 starts sensing even before $t = 16$ seconds. Moreover, this experimentally validates the AFV framework functionality with real devices.

We further consider user activity (i.e., walking or standing) as another example of context. We leverage the step detector in the smartphone to detect changes in the user activity. Figure 6(b) shows that when context changes from standing to walking, the change in the user activity is detected by the *Context Monitoring* and the accelerometer sensing function switches to the smartphone. A detailed description of this experiment and its results are reported in the following section.



(a) Energy consumption. (b) Network throughput. (c) Monetary cost of bandwidth.

Figure 7. Benefits to the user under different system objectives.

User Benefits

We now present experimental results to show quantitative user benefits due to the use of AFV. We developed a fitness tracking application requiring accelerometer functionality and data upload to Internet servers that is similar to the applications listed in Table 1. The application is installed in the LG Nexus 5 smartphone and has the counterpart installed in the LG Urbane smartwatch.

We use four scenarios to show the benefit to the user by adopting AFV. In these scenarios, we set four different objectives considering (1) Information quality; (2) Device's lifetime; (3) Network throughput; and (4) Monetary cost. The considered context information are depending on the relevant objective. The first 2 scenarios use the fitness application and the last 2 scenarios consider a file transfer application.

(1) Information quality. We consider how user activity affects the obtained quality of information. The user is wearing the smartwatch and has the smartphone nearby. At first, the user is standing while doing fitness exercises. The smartwatch is performing accelerometer sensing as there is no activity detected by the smartphone (using step detector) which is on the table. After a while, the user starts walking, carrying the smartphone in their pocket. The smartphone detects activity and the sensing function moves from the smartwatch to the smartphone as the smartphone is chosen in the user preferences for the "walking" context. In order to avoid unnecessary functionality movements between devices for this context, AFV triggers a function placement change only if the new activity continues for at least 20 seconds. Figure 6(b) shows the accelerometer data of the two devices during the experiment.

(2) Device lifetime. Our attempt is to maintain the state of charge (SoC), i.e. the remaining battery capacity, of the devices as long as possible. If SoC drops below a certain threshold (20% for this experiment), a context change will be triggered. We consider a scenario that the smartwatch is preferred for walking activity in the user preferences and performing the accelerometer sensing at the fastest speed and transmitting data once a minute to the smartphone. When SoC of smartwatch goes below threshold, AFV instructs the devices to switch the sensing function to the smartphone with a higher SoC. Figure 7(a) shows results for the increased longevity of the smartwatch battery while using AFV. The uptime of the smartwatch is increased by 5 hours due to the offloading of sensing function to the smartphone below threshold, which is a significant amount of time that may allow the device to reach the next charging cycle.

(3) Network throughput. For this use case, the objective of the AFV framework is to automatically accesses the highest quality network in a wireless environment. The quality is determined by the network throughput and the function of interest is uploading a file of sensor data to an external server for processing. We created two WiFi networks with different throughputs to emulate the heterogenous network. At first, the smartphone is connected to the lower throughput network and the smartwatch has no connectivity. After a while, the smartwatch recognizes a higher throughput network. The availability of a higher throughput network triggers a context change which invokes the *Decision Engine* to select the higher throughput network to upload the file. The achieved throughput is measured at the access points by using a network analyzer (Wireshark[8]). Figure 7(b) depicts the throughput of the device for a periodic upload of 6MB of data before and after context changes. The system detects the availability of the higher quality network and automatically switches to that network in less than 1 minute without any user interaction.

(4) Monetary cost of data usage. Our objective is to manage the capped data plans of the devices belonging to the same PAN. The cost of each network has to be preconfigured with the framework. For this example scenario, we defined that high cost plan costs $0.10/MB and low cost plan costs $0.05/MB. As in the previous case, the availability of a lower cost network connection triggers a context change and AFV notifies the device connected to that network to take over the connectivity function. Figure 7(c) shows the monetary cost for two 6MB data uploads with and without AFV.

RELATED WORK

Adaptive wearable applications have been in existence for nearly 20 years [19]. Early work provided prototype implementations [10], programming language support for existing applications [12], and architectures for system design [5, 23]. Despite this work, applications on commercially available devices have only recently been deployed, due to the challenges of battery and device form, among other issues [3, 15, 21].

Our work follows the philosophies initiated in the early design work on wearables. In particular, Speakeasy [5] motivates the need for domain independent interfaces, mobile code, and user interpretation of semantics. Our representation of context is similar to that provided by Speakeasy and we retain user discernment as well. We are less ambitious in the overall goals as we leverage existing APIs and focus on the adaptive nature of wearable network applications. We do not implement code migration; we provide a system-level extension

[8] https://www.wireshark.org/

to code already available on wearable devices. Smailagic and Sieworek [23] provide key design principles/challenges for future wearable applications: user interface models, input/output modalities, matched capability with requirements, and quick interface evaluation methodology. We focus on the third of these challenges to meet the user's needs with the lowest resource utilization.

CAreDroid [6] is a framework in which to design Android context-aware applications to select the most appropriate functions to run for a given application on a single device. It takes care of context-monitoring and adaptation decisions and allows the developer to focus on application logic only. Their work provided the inspiration for our focus on distributed system applications for wearable computer network applications. AFV differs in that it provides seamless function placement across devices of a PAN and function sharing across applications. Our optimization engine runs as a lightweight separate process on Tier 1 devices and the adaptation selects which functions from which devices are active at any point in time and what communication strategy will be deployed.

Senergy reduces the energy usage of the sensing activity without requiring programmer intervention via the Latency, Accuracy Battery (LAB) abstraction [9]. These 3 components are the main considerations in our framework as well, as they provide a meaningful set of tradeoffs for the user and the developer. The authors develop classifiers to infer context in sensing applications, while we use a simpler sensing strategy, but provide adaptation to achieve application goals. Code In The Air [7, 20] provides a framework designed to enable developers to create tasks for multiple devices and/or multiple devices as well as enable users to specify logical conditions related to the context of the devices participating in the application. If appropriate, code is partitioned and executed on different devices, dependent on the organization of the tasks and with which devices they are associated.

There has been substantial work on context awareness and sensing for mobile apps. Always-on sensing can quickly drain battery resources [22], yet continuous context monitoring is essential for proper response to context changes [8]. This suggests a distinction between always-on and continuous that does not degrade application adaptivity nor battery life. These tradeoffs are explored in several research projects.

The most comprehensive sensing framework is SeeMon [8]. Their approach leverages the relationship between sensor values and higher level "context" states to minimize the number of sensors and their associated energy costs while continuously recognizing context changes. Another approach to sensing is to use a low-powered sensor processor to save energy. MobileHub [22] provides a framework that determined optimized alerts and submission of sensor data that reduce energy without affecting application semantics. Our context has a limited number of sensors and a small number of devices capable of performing context recognition, so we have simplified the evaluation of sensor readings with a call-back mechanism for each activated sensor to inform the smart device regarding changes to the value of interest.

Implementations of mechanisms for energy-efficiency through dynamic assignment of functionality inside the Operating System are achieved in ErdOS [24] and OSone [17]. ErdOS leverages resources in nearby devices based on user modeling and stated user preferences. It uses a lightweight IPC and network stack to securely broadcast important context information and application data in a user-level communication manager. This could be extended to communications with low-powered sensor devices. OSone distributes the functionality of the operating system in a similar fashion to how Barrelfish [1] separates functionality onto different cores. The architecture consists of a kernel node in charge of various host nodes that can be kept simpler. We implement AFV in a similar fashion with the potential to have multiple controlling nodes over time, depending on remaining resources and application needs.

Wireless Body Area Networks have explored various scenarios for applications, from medical and health monitoring to entertainment and fitness applications, on both wearable and implant devices [16]. There are also implementations of wearable smart textiles [18]. The focus in this community has been on the lower levels of network communication both intra-WBAN (on a person) and inter-WBAN (between individuals). Our work is at a higher level and can leverage optimizations made in the network and physical layer transmissions.

DISCUSSION AND CONCLUSION

In this paper, we proposed AFV, a framework providing context-aware application function virtualization on a personal area network comprised of multiple wearable devices. We designed a set of APIs that can be easily leveraged by developers to design context-aware wearable applications, hiding the optimal function allocation algorithm from the developers.

AFV provides significant benefits to application developers and end users which can optimally leverage the functionalities available in the PAN without having any prior knowledge about the PAN and without development and configuration effort. For instance, our simulation results show that AFV increases the system uptime up to 35-40% compared with typical configurations of current PAN applications. We experimentally validated the framework by implementing AFV on Android devices and experiments on a set of significant use cases. The overhead of function allocation algorithm is a few milliseconds for realistic scenarios and adaptation to context changes can be performed in less than one second with real devices.

Current AFV function placement may lead to inefficiency for some metrics in some scenarios. As the function allocation algorithm is designed to minimize a given set of costs while respecting user preferences and contexts, other non-optimized metrics may be affected with respect to a vanilla scenario without AFV. We plan to address such inefficiencies as future work by designing a more complex function allocation algorithm taking into account multiple metrics and improving the *Decision Engine* to take personalized optimal decisions involving machine learning algorithms. We will also develop the AFV implementation with awareness to more context changes which would lead to much better proactive decision making capabilities.

REFERENCES

1. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proc. SOSP*. ACM, New York, NY, USA, 29–44.
2. A. Beach, M. Gartrell, X. Xing, R. Han, Q. Lv, S. Mishra, and K. Seada. 2010. Fusing Mobile, Sensor, and Social Data to Fully Enable Context-aware Computing. In *Proc. HotMobile*. ACM, New York, NY, USA, 60–65.
3. S. Brachmann. 2014. Wearable Gadgets: What is the Secret to Commercial Success? (2014).
4. J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne. 2016. Characterization of Early Smartwatch Apps. In *Proc. WristSense Workshop*. IEEE, Piscataway, NJ, USA, 598–603.
5. W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. 2002. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proc. Mobicom*. ACM, New York, NY, USA, 279–286.
6. S. Elmalaki, L. Wanner, and M. Srivastava. 2015. CAreDroid: Adaptation Framework for Android Context-Aware Applications. In *Proc. MobiCom*. ACM, New York, NY, USA, 386–399.
7. T. Kaler, J. P. Lynch, T. Peng, L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. 2010. Code in the Air: Simplifying Sensing on Smartphones. In *Proc. SenSys*. ACM, New York, NY, USA, 407–408.
8. S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. 2008. SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *Proc. MobiSys*. ACM, New York, NY, USA, 267–280.
9. A. Kansal, S. Saponas, A.J. Bernheim Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. 2013. The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing. In *Proc. OOPSLA*. ACM, New York, NY, USA, 661–676.
10. G. Kortuem, Z. Segall, and M. Bauer. 1998. Context-aware, Adaptive Wearable Computers as Remote Interfaces to 'Intelligent' Environments. In *Proc, ISWC*. ACM, New York, NY, USA, 58–65.
11. H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. 2010. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *Proc. SenSys*. ACM, New York, NY, USA, 71–84.
12. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and R. Kalaskar. 2002. Programming Language Support for Adaptable Wearable Computing. In *Proc. ISWC*. ACM, New York, NY, USA, 205–212.
13. R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. 2016. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials* 18, 1 (2016), 236–262.
14. C. Min, S. Kang, C. Yoo, J. Cha, S. Choi, Y. Oh, and J. Song. 2015. Exploring Current Practices for Battery Use and Management of Smartwatches. In *Proc. ISWC*. ACM, New York, NY, USA, 11–18.
15. Johanna Mischke. 2014. Wearables for Professional and Industry Applications. (2014).
16. S. Movassaghi, M. Abolhasan, J. Lipman, D. Smith, and A. Jamalipour. 2014. Wireless Body Area Networks: A Survey. *IEEE Communications Surveys Tutorials* 16, 3 (March 2014), 1658–1686.
17. B. Pasztor and P. Hui. 2013. OSone: A Distributed Operating System for Energy Efficient Sensor Network. In *Proc. 25th International Teletraffic Congress (ITC)*. IEEE, Piscataway, NJ, USA, 1–9.
18. V. Peiris. 2013. Highly integrated wireless sensing for body area network applications. (2013).
19. Cliff Randell. 2005. *Wearable Computing: A Review*. Technical Report Technical Report CSTR-06-004. University of Bristol.
20. L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. 2012. Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones. In *Proc. MobiSys*. ACM, New York, NY, USA, 4:1–4:6.
21. R. Rawassizadeh, B. A. Price, and M. Petre. 2014. Wearables: Has the Age of Smartwatches Finally Arrived? *Commun. ACM* 58, 1 (Dec. 2014), 45–47.
22. H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. 2015. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proc. UbiComp*. ACM, New York, NY, USA, 227–238.
23. A. Smailagic and D. Siewiorek. 2002. Application Design for Wearable and Context-Aware Computers. *IEEE Pervasive Computing* 1, 4 (Oct. 2002), 20–29.
24. N. Vallina-Rodriguez and J. Crowcroft. 2011. ErdOS: Achieving Energy Savings in Mobile OS. In *Proc. MobiArch*. ACM, New York, NY, USA, 37–42.
25. D. P. Williamson and D. B. Shmoys. 2011. *The Design of Approximation Algorithms* (1st ed.). Cambridge University Press, New York, NY, USA.
26. Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Ylä-Jääski, and S. Tarkoma. 2014. Modeling Energy Consumption of Data Transmission Over Wi-Fi. *IEEE Transactions on Mobile Computing* 13, 8 (Aug. 2014), 1760–1773.