SensorLib: an Energy-efficient Sensor-collection Library for Wear OS

Lorenzo Calisti
l.calisti@campus.uniurb.it
Department of Pure and Applied Sciences University of Urbino
Urbino, Italy

Emanuele Lattanzi
emanuele.lattanzi@uniurb.it
Department of Pure and Applied Sciences University of Urbino
Urbino, Italy

ABSTRACT

In recent years, wearable technology has gained popularity due to features like long battery life, network connectivity, and fitness monitoring. Human Activity Recognition has emerged as a popular use case for smartwatches, enabling the recognition of activities starting from internal sensors. Data acquisition from sensors is crucial in wearable devices because if not properly implemented can reduce battery life or device responsiveness. The paper presents an energy-efficient programming library for real-time sensor sampling on smartwatches using native Wear OS sensor APIs. The library's implementation is evaluated on a real smartwatch for code size, memory utilization, and power consumption. The preliminary results empirically demonstrate that the solution proved to be light and versatile enough to be used on wearable devices without heavily compromising battery life and system performance.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; • **Human-centered computing** → **Empirical studies in ubiquitous and mobile computing**.

KEYWORDS

energy-efficient programming, wearable devices, human activity recognition, Wear OS

ACM Reference Format:
Lorenzo Calisti and Emanuele Lattanzi. 2023. SensorLib: an Energy-efficient Sensor-collection Library for Wear OS. In *The 4th European Symposium on Software Engineering (ESSE 2023), December 01–03, 2023, Napoli, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3651640.3651641

1 INTRODUCTION

In recent years, wearable technology has become increasingly popular. Wearable devices have gone from simple futuristic gadgets for the niche to common tools used by the masses. Among the causes of this rise in popularity, we certainly find the presence of a rich range of features such as long battery life, internet connectivity, media management, ability to reply to messages by voice and receive notifications without having to interact with your smartphone. In addition to this, the presence of integrated sensors for automatic monitoring of fitness activities, combined with the increase in health awareness, are aspects that will lead to a sure increase in the smartwatch market [16].

Modern chip manufacturing techniques allow the creation of smartwatches equipped with several GB of RAM and the adoption of multi-core processors. The computing power of these devices is comparable to that of the top-of-the-range smartphones of a few years ago. Nowadays smartwatches mainly serve the role of smartphone extension by connecting to it via technologies such as Bluetooth LE or WiFi. The strength of wearable devices is the presence of various sensors for movement, position, and environment, such as GPS, accelerometer, gyroscope, heart-rate sensor, blood oxygenation sensor, and step counter.

One of the uses of smartwatches that is becoming very popular is Human Activity Recognition (HAR). HAR plays an important role in human-to-human interactions allowing one to obtain information about a person's identity, personality, and psychological state. Activity recognition is not a daunting task for humans, but it can be extremely complex for an algorithm. In recent years, the field of Human Activity Recognition has expanded to include techniques such as computer vision and machine learning that facilitate the process of recognizing activities and make it more precise even for a machine[15, 21]. HAR can be divided into two macro-categories: (i) vision-based, (ii) sensor-based. In the former, the activities are obtained starting from video and using computer vision, while in the latter the data is extracted from different sensors such as accelerometer and gyroscope[7].

In the context of HAR, one of the most critical factors is the acquisition of data produced by inertial sensors such as accelerometers and gyroscopes which are time-dependent; the samples cannot be taken individually but must be considered in temporal order. Furthermore, these types of sensors sample data very quickly, reaching frequencies greater than 100 Hz. When dealing with dense streams of data, a technique widely used to extract relevant information is that of windowing. Time windowing consists in dividing the collected data into equal time periods in each of which the sampling frequency is kept constant [18].

Modern wearable operating systems, such as Wear OS or Apple Watch OS, for instance, provide developers with access to general-purpose APIs dedicated to collecting data from sensors installed on the device. Unfortunately, these APIs do not directly sample using time windowing, but use the callback mechanism to notify the requesting application of each new sample as soon it is available. Consequently, anyone who needs to work with windowed data

ESSE 2023, December 01–03, 2023, Napoli, Italy

ACM ISBN 979-8-4007-0881-7/23/12
https://doi.org/10.1145/3651640.3651641

is forced to re-implement the windowing mechanism around the native API.

Beyond having to rewrite an ad-hoc library, the main issue is that an incorrect or hasty implementation could negatively impact performance by reducing, for example, battery life or device responsiveness. In fact, one of the major challenges associated with wearable devices is their limited battery life, which hinders their continuous and prolonged usage. For these reasons, writing energy-efficient code plays a crucial role in addressing this challenge, ensuring prolonged operation and user satisfaction [13, 14].

In this paper, we present the architecture and the implementation of an energy-efficient programming library, for collecting data from internal sensors, focused on real-time windowed data sampling that leverages the native Wear OS sensor APIs. The library has been implemented and tested on a real smartwatch to deeply characterize its runtime behavior in terms of code size, memory utilization, and power consumption by means of an experimental measurement setup.

The rest of this paper is organized as follows: in Section 2 we analyze the current scientific literature in the field of energy-efficient sensor-based collection libraries; in Section 3 we present a background description of the Wear OS system and the APIs available to access key system functionalities; in Section 4 we describe the proposed architectures and its implementation; In Section 5 we describe the measurement setup; in Section 6 we report the preliminary characterization results and, finally, in Section 7 we summarize the main contributions and findings of this work and report some future directions.

2 RELATED WORK

The problem of the energy consumption of the software devoted to wearable devices has been the subject of many studies in recent years; research in this field can be divided into two main strands: (i) energy efficiency of Android devices, (ii) energy consumption of wearables.

Many studies consist of a comparison of the current literature in the field of energy efficiency of Android devices. For example, L. Cruz et al. analyzed commits, issues, and pull requests of 1,021 Android applications and 756 iOS applications to catalog 22 design patterns showing a more significant effort in improving energy efficiency made by Android developers (25%) rather than iOS developers (10%) [6]. However, Ullah et al. in 2022 list various problems and solutions by analyzing 19 studies in which it emerges that bad programming practices are the most discussed problem in literature (26%). In contrast, problems related to tools and design patterns are less discussed (15.7%) [20].

L. Corbalan et al. presented a comparison of the impact that native programming and cross-platform programming have on the energy consumption of applications analyzing three types of apps and seven different development frameworks [4]. Moreover, S. Huber et al. have compared the efficiency of Progressive Web Apps (PWA) with other development approaches noting how PWAs can be a valid alternative to cross-platform systems, while native development remains always the best approach[10].

A part of the current research works focuses on the link between the so-called "code smells" and the increase in energy consumption. For instance, in 2019 Palomba et al. show that not all "smells" have the same impact on energy efficiency [17], while L.Cruz et al. investigated 8 "code smells" associated with performance loss including ViewHolder, DrawAllocation, WakeLock, ObsoleteLayoutParam, and Recycle highlighting the need for a refactoring tool that keeps these common errors in mind [5].

Concerning wearable devices such as smartwatches, one of the major contributions has been done by H. Zhang et al. in 2018. Here the authors have developed a control model capable of detecting patterns of energy inefficiency for sensors and displays on Android Wear [22]. Moreover, J. Kim et al. demonstrate how storage IO activities contribute in a non-trivial part to the power consumption of wearable devices; for this reason, they have developed a set of techniques such as Metadata Embedding, Selective Directory Sync, and Flushless Durability Guarantee able to improve battery life by 3% [11].

Although the energy consumption of wearable software is a well-known and studied research field, to the best of our knowledge, we are the first researchers presenting and characterizing an energy-efficient user-level library aimed at collecting sensor data in real-time in Wear OS.

3 BACKGROUND

In this section, we provide an overview of the different technologies involved in the development of sensor-based HAR applications on top of the Wear OS operating system.

3.1 Wear OS

Wear OS, formerly known as Android Wear, is a version of Google's Android operating system designed specifically for smartwatches and optimized for the wrist[1]. The platform was announced in 2014 and supports connectivity technologies such as Bluetooth, NFC, Wi-Fi, and LTE. The system integrates many of Google's own services such as voice assistant, Google Maps, Gmail, and Google Wallet, it also allows you to download specially created third-party apps from the Google Play Store.

The development of a Wear OS application is very similar to that of a normal Android smartphone application using the Android Software Development Kit (Android SDK) for which most of the APIs and available features are in common, except for some specific to Wear OS. Traditionally Android application development is done using the Java programming language which, once compiled into bytecode, is packaged into an Application Package Kit (APK) which can be published within the Google Play Store.

In 2017, Google officially announced the support for the Kotlin programming language that combines object-oriented functionality with functional programming features within a pragmatic statically typed language. Kotlin greatly reduces the verbosity of Java and increases its security thanks to constructs such as higher-order functions, extension functions, null-safety, data classes, immutable array types, and coroutines [9].

3.2 Kotlin Coroutines, Channels, and Flow

Although they were invented more than 50 years ago, there is no precise definition of what coroutines are. When we talk about coroutines we generally mean a function whose execution can be

suspended and resumed preserving its state. Definitely, coroutines are very similar to threads, however, the former operates following cooperative multitasking, while the latter uses preemptive multitasking. Coroutines offer several advantages over threads. First, they are simpler to use since they don't require complex synchronization primitives such as Semaphores or Mutexes that make the code error-prone. The second advantage is the absence of system calls or blocking calls which slow down context switching.

In Kotlin, coroutines are built around the concept of suspending function. Suspending functions are declared by adding the "suspend" keyword and are called exactly like normal synchronous functions. Most of the features related to coroutines are not native to the language but are implemented directly in Kotlin as part of the standard library. During compilation, each suspending function is transformed into a Continuation-Passing Style function (CPS) in which there is an additional parameter of type Continuation[8]. Due to some limitations of the JVM, suspending functions are implemented as state machines where Continuation objects maintain the state of local variables during the suspension.

Another difference between programming with threads and coroutines is the way processes communicate with each other. Traditionally, asynchronous programming through Threads is based on the concept of Shared Memory in which the programmer manages the synchronization and protection of memory accesses, while the Coroutines use Message Passing techniques in which synchronization is implicit by means of objects called Channels [12].

The Flow object was recently introduced to Kotlin with the goal of managing and observing asynchronous data flows in a simple way. The Flow class is built on top of the Coroutines and conceptually behaves like a stream that computes and outputs multiple values sequentially. A Flow is very similar to an iterator but makes use of the suspend functions to block the producer and the consumer in order to obtain an asynchronous structure[2].

The three fundamental elements of Flow are producers, intermediaries, and consumers. Producers output values continuously in the data stream. Intermediaries are able to modify each value of the stream and then re-emit it; this way, it is possible to filter or transform the data produced transparently to the consumers. Consumers use the stream of data applying operations called terminals such as `collect`, `toList`, or `first`.

3.3 Sensor framework

Concerning the access and use of the sensors present in Wear OS, there are no substantial changes, with respect to the Android framework, at the system APIs charge other than the addition of new types of sensor present only in a watch context. For this reason, the following description of the Android Sensor Framework can be considered both for traditional Android development and for Wear OS development.

Most devices powered by the Android operating system have some kind of built-in sensor able to provide raw data in real-time with great precision and accuracy. They are helpful in monitoring three-dimensional device movement and positioning or monitoring changes in the ambient environment near a device [3]. The Android platform supports three categories of sensors: i) motion sensors; ii) environmental sensors; iii) position sensors. Motion sensors measure acceleration and rotational forces along three axes; this category includes accelerometers, gravity sensors, and gyroscopes. Environmental sensors measure various environmental parameters such as ambient air temperature, illumination, and humidity; this category contains barometers, photometers, and thermometers. Position sensors measure a device's physical position; this category contains orientation sensors and magnetometers.

To access the sensors available on a device Android provides the Sensor Stack. The topmost element of the stack is the application requesting data from one or more sensors; to do so, the application makes calls to the Android SDK which, in turn, are redirected to the internal Framework. The Framework is the third element of the stack and has the purpose of receiving and managing all requests from applications. When two applications request access to the same sensor at the same time, the Framework multiplexes the requests and uses the highest sampling frequency among those required. This means that when an application requests data at a particular rate, it is not guaranteed to always get it at that frequency, but the data may arrive faster. At the bottom of the stack are the drivers and the sensor HUBs, these elements are managed by the device manufacturers and communicate with the Framework through the Hardware Abstraction Layer (HAL). The last element of the stack is composed of the physical sensors that measure the environmental data.

For what concerns the application developers the Android SDK is the main access point to the sensor stack; it contains several classes that help perform a wide variety of sensor-related tasks, such as determining which sensors are available or acquiring raw sensor data. The most important classes and interfaces inside the Android Sensor Framework are the following:

- The `SensorManager` class provides various methods for accessing sensors and registering or unregistering sensor event listeners.
- The `Sensor` class offers all the information related to a specific sensor and provides methods to determine its capabilities.
- The `SensorEvent` object represents a single event of a sensor and contains information such as the type of sensor that produced it, the timestamp of the event, the accuracy, and the raw data.
- The `SensorEventListener` is an interface that acts as a callback to receive sensor events from the system.

Normally the SensorEventListener callback is executed on the application's main thread together with all the code responsible for updating the UI and interacting with the system components, for this reason, it is possible to pass to the registerListener method a `Handler` associated with a different thread onto which redirect the event handling; in this way, slowdowns in the UI are avoided and the overall user experience is improved.

Sensors can produce events in different ways depending on the type of sensor we are considering. *Continuous sensors* produce events at a fixed rate from the time the SensorEventListener is registered until it is unregistered; most of the common sensor types such as accelerometer, gyroscope, and magnetometer belong to this category. *On-change sensors* produce events only if the measured

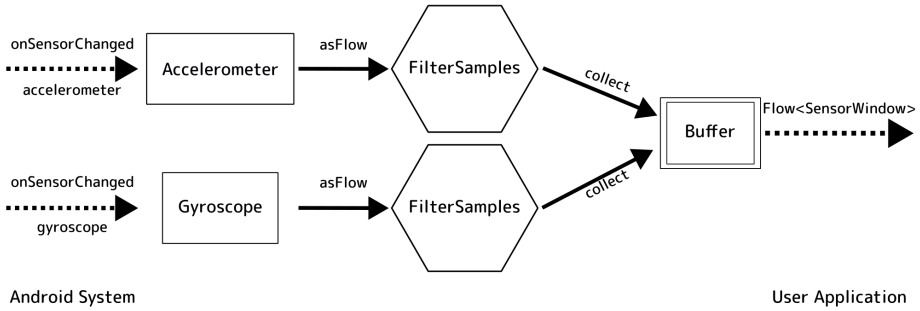

Figure 1: Schematic diagram of the library implementation.

value undergoes a change since the last time it was measured, sensors such as the pedometer, ambient pressure sensor, and heart rate sensor belong to this category. *One-shot sensors*, on the other hand, produce a single event after which they unregister themselves and to be used again a new listener must be registered. Significant motion and wake-up gesture sensors belong to this category.

4 THE PROPOSED LIBRARY

In this paper, we present the development of a library for Wear OS that aids in the real-time collection of data from sensors at a constant sample rate and with data grouped into windows of fixed size. This library is called *Sensorlib*. The library allows the user to configure the sampling period in milliseconds and the window size; these two parameters alone are enough to change the sampling rate and the duration of the window in seconds.

The library is designed to work with any sensor available on the Android device. Still, since the data is windowed it is better suited to work with continuous sensors type. The accelerometer and gyroscope sensors have been chosen mainly to show the ability of the library to work with multiple sensors simultaneously and because these two types of sensors are the most popular for the HAR applications this library is oriented towards.

4.1 Implementation

To implement the proposed library we make use of `Flow` and `Channel` classes. Figure 1 represents the schematic diagram of the library which highlights data collection phases. In particular, two flow instances (one for each sensor) receive the data from the Android system, which, after being filtered separately, are merged into a single channel that triggers the user application when full.

The main classes and methods of this version are:

- `SensorFlow`
- `SensorDataListener`
- `Flow.filterSamples`
- `WindowBuffer`

The `SensorDataListener` class collects events from a single sensor specified in its constructor. The collection implements a `SensorEventListener` in a special thread named after the sensor type and passes all the events directly to a Flow without any processing.

Event filtering is done by the extension function `Flow.filterSample` which plays the intermediary role by taking a `Flow` with data from a sensor and returning a new one by dropping unnecessary data.

The `WindowBuffer` class wraps the window creation logic into a single helper class. His `appendSample` method takes care of inserting a new event in the correct place and returns an enum signaling that window is full.

Listing 1 reports the code of the function `SensorFlow.asFlow` which is the library entry point. This function internally calls the `channelFlow` method which allows for the creation of a Flow passing the data to an internal Channel. The `launch` function is used to spawn a new Coroutine with the specified code. In this method three different Coroutines are created: the first two collect the data respectively from the accelerometer and gyroscope, filter them and finally apply the terminal function `collect` which sends every event in a shared Channel. The last Coroutine listens to these events, appends them to the `WindowBuffer`, and sends the produced windows to the returned Flow.

```
fun asFlow(): Flow<SensorWindow> =
  channelFlow {
      launch {
          accListener.asFlow()
              .filterSamples(samplingMs)
              .collect { sensorChannel.send(it) }
      }
      launch {
          gyrListener.asFlow()
              .filterSamples(samplingMs)
              .collect { sensorChannel.send(it) }
      }
      launch {
          for (event in sensorChannel) {
              if (windowBuffer.appendSample(event) ==
                  WindowBuffer.Status.Full) {
                  this@channelFlow.send(windowBuffer.window)
              }
          }
      }
  }

```

Listing 1: Code of the library entry point function

5 EXPERIMENTAL SETUP

We used an OPPO Watch 46mm equipped with a Qualcomm ® Snapdragon Wear™ 3100 as a reference device. It provides 1 GB of

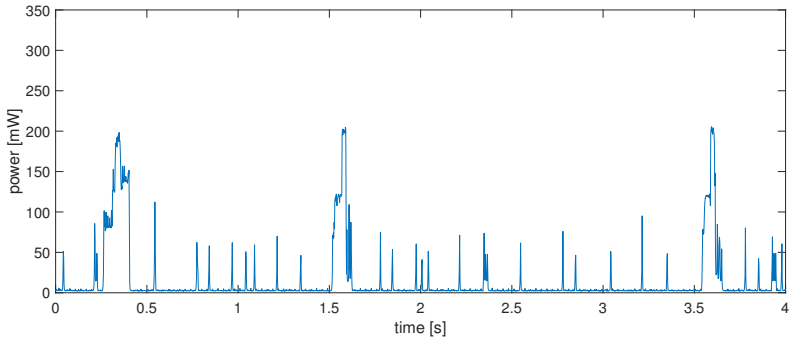

Figure 2: A power consumption trace of the proposed library implementation during continuous accelerometer and gyroscope data collection.

ram and 8 GB of non-volatile flash memory and it is also equipped with several sensors such as an accelerometer, gyroscope, magnetic field sensor, barometer, etc. The whole device is powered by means of a 430 mAh Li-Po battery that ensures up to 36 hours of autonomy. Concerning the software, the device runs Wear OS by Google which allows a high degree of programmability.

To measure the energy consumption, the smartwatch has been opened, the battery has been removed and it has been powered at 3.85V through an NGMO2 Rohde & Schwarz dual-channel power supply [19]. The energy consumption has been measured by recording, at a frequency of 10 kHz, the voltage drop across a sensing resistor (1.5Ω) placed in series with the power supply of the device.

6 CHARACTERIZATION RESULTS

In order to characterize the proposed implementation of the sensor collection library, we create a dummy application that uses it by collecting data at a fixed sampling size and over a predefined time window.

Table 1: Characterization results of the proposed library implementation.

	value	standard deviation
file [#]	8	-
lines of code [#]	316	-
objects allocation [#/s]	690	184
heap size [kB]	597	49
average power [mW]	32	3.6

Table 1 reports the results of the deep characterization of the proposed library from several points of view. First of all, we measured the total number of files and lines of code composing our implementation to evaluate the compactness of the code, then we monitor ten different executions lasting 5 minutes each to collect energy consumption traces, by means of the experimental setup described in Section 5 together with memory usage statistics using the integrated Android Studio profiler.

From the code analysis point of view, the proposed implementation results in a very compact code, counting only 316 lines in total distributed over 8 files. The compactness of the code is also reflected in the amount of memory used at runtime which does not exceed, on average, 600 kB.

A separate discussion must be made for the number of new objects allocated per second which, on the other hand, may seem quite high, settling on the value of about 690 ± 184. In fact, a large number of dynamically allocated and de-allocated objects (the size of the heap remains constant at about 600 kB throughout the execution) must be seen from the point of view of the object-oriented programming paradigm which, by its nature, makes use of many small size objects. On the other hand, considering the very low average power consumption, which is around only 32 ± 3.6 mW, the rapid objects allocation and de-allocation does not seem to impair the energy consumption.

Notice that, in our experiments, the application does not perform any type of processing on the data except for copying to a buffer so as not to influence the characterization of the library operation. Moreover, as we measure the real energy consumed by the whole device, we disabled all the connection interfaces and we collected the energy data over long runs to minimize the possible interference due to the periodic execution of some system tasks.

To better understand how the proposed library takes advantage of the device's hardware and how this affects the energy absorbed, we reported, in Figure 2, a power consumption trace recorded during continuous accelerometer and gyroscope data collection at 50 Hz over a time window of two seconds.

Analyzing the trace from a qualitative point of view, it is possible to notice some phases of intense power consumption alternate with phases in which this remains close to zero. In the latter, the system's power management presumably manages to disable most of the electronic circuits in order to save energy while the former represents bursts of active computation. In general, it seems evident that, although a quite large number of allocations and de-allocations per second, the smartwatch hardware remains in low-energy states most of the time allowing you to save energy.

6.1 Sensitivity to the sampling parameters

The sensitivity to the two parameters that influence the functioning of the software (sampling rate and window size) was evaluated by

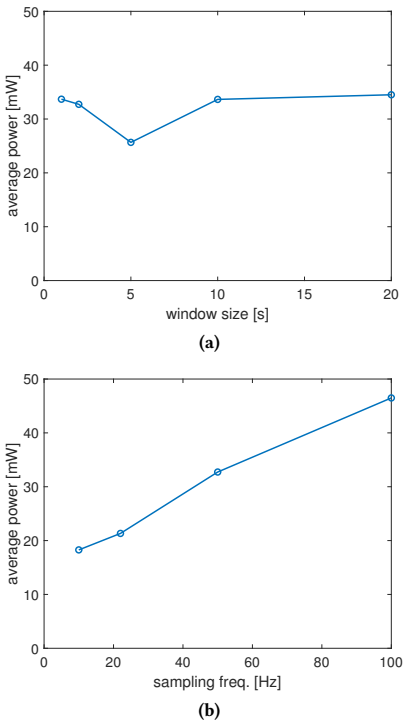

Figure 3: The average power consumption of the proposed library when varying the size of the window processing (a) and the sampling frequency (b).

repeating the experiments and varying one of the parameters at a time.

Figure 3 shows the average power consumption of the proposed library when varying the size of the processing window (a) and the sampling frequency (b). Concerning the size of the processing window, there is no evidence of an influence of this on the energy consumed. This behavior appears plausible considering that the size of the window essentially influences the operation of the final stage of the library, i.e. the callback to the user application which, in all the experiments, performs nothing more than a copy of the data. Notice that, in these experiments, the sampling frequency was set to 50 Hz.

On the contrary, increasing the sampling frequency, Figure 3.(b), clearly seems to lead to an increase in the average power consumption, starting from about 18 mW for a sampling frequency set to 10 Hz, and reaching the maximum value of about 46 mW for a frequency of 100 Hz. In this case, the size of the processing windows was 2 seconds. This means that, by opportunely tuning the sampling parameters, given the factory battery of about 480 mAh, the proposed library can continuously run for up to 24 hours.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented an energy-efficient programming library for collecting data from internal sensors in the Wear OS ecosystem. The library has been deeply characterized in terms of code size, memory utilization, and power consumption on a real smartwatch by means of an experimental measurement setup. The preliminary results empirically demonstrate that the solution proved to be light and versatile enough to be used on wearable devices without heavily compromising battery life and system performance.

In the future, we planned to make a performance comparison with other implementations which make use of different technology solutions such as for instance, Java threads or pure Kotlin Flows.

REFERENCES

[1] Android. 2023. Build apps for the wrist with Wear OS. https://developer.android.com/wear?hl=it Online accessed 28 April 2023.
[2] Android. 2023. Kotlin flows on Android. https://developer.android.com/kotlin/flow?hl=en Online accessed 16 June 2023.
[3] Android. 2023. Sensors Overview. "https://developer.android.com/guide/" Online accessed 21 February 2023.
[4] Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. 2018. Development frameworks for mobile devices: a comparative study about energy consumption. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 191–201.
[5] Luis Cruz and Rui Abreu. 2017. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 46–57.
[6] Luis Cruz and Rui Abreu. 2019. Catalog of energy patterns for mobile applications. *Empirical Software Engineering* 24 (2019), 2209–2235.
[7] L Minh Dang, Kyungbok Min, Hanxiang Wang, Md Jalil Piran, Cheol Hee Lee, and Hyeonjoon Moon. 2020. Sensor-based and vision-based human activity recognition: A comprehensive survey. *Pattern Recognition* 108 (2020), 107561.
[8] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 68–84.
[9] Bruno Góis Mateus and Matias Martinez. 2019. An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering* 24 (2019). https://doi.org/10.1007/s10664-019-09727-4
[10] Stefan Huber, Lukas Demetz, and Michael Felderer. 2022. A comparative study on the energy consumption of Progressive Web Apps. *Information Systems* 108 (2022), 102017.
[11] Junghoon Kim, Sundoo Kim, Juseong Yun, and Youjip Won. 2019. Energy efficient IO stack design for wearable device. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2152–2159.
[12] Nikita Koval, Dan Alistarh, and Roman Elizarov. 2023. Fast and Scalable Channels in Kotlin Coroutines. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 107–118.
[13] Emanuele Lattanzi, Lorenzo Calisti, et al. 2023. Energy-aware Tiny Machine Learning for Sensor-based Hand-washing Recognition. In *International Conference on Machine Learning Technologies (ICMLT 2023)*. ACM.
[14] Emanuele Lattanzi, Matteo Donati, and Valerio Freschi. 2022. Exploring Artificial Neural Networks Efficiency in Tiny Wearable Devices for Human Activity Recognition. *Sensors* 22, 7 (2022), 2637.
[15] Maxim Maximov, Sang-Rok Oh, and Myong Soo Park. 2017. Real-Time Action Recognition System from a First-Person View Video Stream. *International Journal of Computer Theory and Engineering* 9, 2 (2017), 79–86.
[16] Sanjay M Mishra. 2015. *Wearable android: android wear and google fit app development*. John Wiley & Sons.
[17] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (2019), 43–55.
[18] Bronagh Quigley, Mark Donnelly, George Moore, and Leo Galway. 2018. A Comparative Analysis of Windowing Approaches in Dense Sensing Environments. *Proceedings* 2 (2018). https://doi.org/10.3390/proceedings2191245
[19] Rohde & Schwarz. 2020. NGMO2 datasheet. Retrieved 2022-07-19 from https://www.rohde-schwarz.com/it/brochure-scheda-tecnica/ngmo2/
[20] Obaid Ullah, Muhammad Hanan, and Maryam Abdul Ghafoor. 2022. Energy Efficiency Issues in Android Application: A Literature Review. In *2022 24th International Multitopic Conference (INMIC)*. IEEE, 1–6.
[21] Michalis Vrigkas, Christophoros Nikou, and Ioannis A. Kakadiaris. 2015. A Review of Human Activity Recognition Methods. *Frontiers in Robotics and AI* 2 (2015). https://doi.org/10.3389/frobt.2015.00028
[22] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2018. Detection of energy inefficiencies in android wear watch faces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 691–702.