

Research article

LwHBench: A low-level hardware component benchmark and dataset for Single Board Computers

Pedro Miguel Sánchez Sánchez^{a,*}, José María Jorquera Valero^a,
Alberto Huertas Celdrán^b, Jérôme Bovet^c, Manuel Gil Pérez^a,
Gregorio Martínez Pérez^a

^a Department of Information and Communications Engineering, University of Murcia, Murcia 30100, Spain

^b Communication Systems Group (CSG), Department of Informatics (IfI), University of Zurich UZH, 8050 Zürich, Switzerland

^c Cyber-Defence Campus within armasuisse Science & Technology, CH—3602 Thun, Switzerland

ARTICLE INFO

Keywords:

Hardware benchmarking

System performance

Dataset

IoT device

Identification

ABSTRACT

In today's computing environment, where Artificial Intelligence (AI) and data processing are moving toward the Internet of Things (IoT) and Edge computing paradigms, benchmarking resource-constrained devices is a critical task to evaluate their suitability and performance. Between the employed devices, Single-Board Computers arise as multi-purpose and affordable systems. The literature has explored Single-Board Computers performance when running high-level benchmarks specialized in particular application scenarios, such as AI or medical applications. However, lower-level benchmarking applications and datasets are needed to enable new Edge-based AI solutions for network, system and service management based on device and component performance, such as individual device identification. Thus, this paper presents LwHBench, a low-level hardware benchmarking application for Single-Board Computers that measures the performance of CPU, GPU, Memory and Storage taking into account the component constraints in these types of devices. LwHBench has been implemented for Raspberry Pi devices and run for 100 days on a set of 45 devices to generate an extensive dataset that allows the usage of AI techniques in scenarios where performance data can help in the device management process. Besides, to demonstrate the inter-scenario capability of the dataset, a series of AI-enabled use cases about device identification and context impact on performance are presented as exploration of the published data. Finally, the benchmark application has been adapted and applied to an agriculture-focused scenario where three RockPro64 devices are present.

1. Introduction

Performance benchmarking has been an issue explored since the early days of computer science [1]. Knowing the capabilities of a device is critical to create applications optimized for it [2]. In this sense, benchmarking has become a priority due to the magnification of the number of online devices provoked by new technologies such as 5G, IoT or cloud. In addition, the explosion of techniques such as Machine Learning (ML) and Deep Learning (DL), which usually require high computational power, has been another key factor increasing the need for processing measurement applications. In this context, *device performance benchmarking* is a research avenue that acquired a large momentum in the last years [3], especially in IoT and Edge computing paradigms.

* Corresponding author.

E-mail address: pedromiguel.sanchez@um.es (P.M. Sánchez Sánchez).

<https://doi.org/10.1016/j.iot.2023.100764>

Received 29 September 2022; Received in revised form 20 February 2023; Accepted 20 March 2023

Available online 3 April 2023

2542-6605/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

Benchmarking can be defined as an intentional stress introduced in a system to measure how the device behaves regarding a determined set of metrics [4]. The main purposes of benchmarking are to (i) verify that the performance of a device is the promised one and suitable for a certain activity such as running AI tasks, and (ii) model the internal behavior of a certain device to identify it or verify that it is running properly [5]. Therefore, device benchmarking can be seen from two perspectives. First, as a high-level system benchmarking that seeks to measure how a certain application works on the device in terms of performance or Quality-of-Service (QoS), for example, in execution time or energy consumption [6]. Second, as a low-level hardware benchmarking, which goal is to characterize the device components in a more precise way [7], so it is possible to differentiate devices or detect imperfections and errors in the chips, among other options. “Low-level” indicates that the focus of the measurements is close to the physical level of the components, measuring values such as frequencies and cycles instead of QoS (time and energy). An example of this type of benchmarking would be to measure how many cycles it takes a processor to execute a certain simple task, measuring whether it meets its specification and the necessary stability requirements.

Moreover, due to the explosion undergone by AI, and more specifically ML and DL [8], these techniques have also landed in the IoT field, being applied in areas such as industry, network and service management or cybersecurity. Thus, the field of IoT benchmarking also has the task of evaluating the performance of training and deploying ML/DL in these devices hardware. Numerous solutions have explored the capabilities of IoT devices, mainly Single-Board Computers (SBC) such as Raspberry Pi, when running different ML/DL algorithms and libraries [9], so this is an area with enough pedigree and many recent works [10]. However, low-level hardware benchmarking and the application of ML/DL in the data generated from these benchmarks remain mostly unexplored in the IoT field. This is an important task as critical functionalities are moving to the IoT and having the components of these devices properly analyzed is essential.

Then, although many benchmarks have been proposed for SBCs in recent years, as [3] shows, some challenges are present in the area, such as (i) many benchmarks are proposed but no exhaustive execution datasets are provided; (ii) all recent benchmarks focus on high-level applications and none of them measures the performance of the hardware from a low-level perspective; (iii) there is no work measuring the components performance from a different component of the same device, which is important to avoid inconsistent values coming from the measured component (e.g. measure an execution time using the CPU as its own source of timestamps), as it cannot notice its own inaccuracy; (iv) only a few solutions consider storage and memory in the benchmark, most of them are focused on CPU QoS (execution time, computation and communication latency); and (v) none of the previous SBC benchmark solutions consider GPU low-level performance for non-graphics processing.

In order to improve the previous limitations and fill the literature gap, the main contributions of the present work are:

- A low-level hardware component benchmarking application, namely LwHBench, which measures CPU, GPU, memory and storage device performance from the device reference point. The benchmark is implemented as a proof of concept for Raspberry Pi devices [11], taking into account the particularities of their hardware components.
- A comprehensive dataset acquired as a result of running the previous benchmark on a set of 45 Raspberry Pi of various models for 100 days [12]. For data collection, a series of measures regarding the stability of the device performance have been taken, setting the frequency of the components to fixed values and trying to reduce as much as possible the possible noise introduced in the measurements by other processes running on the devices. This dataset contains a total of 4 GB of data (2 386 126 vectors), more than any other benchmark dataset in the literature, ready to be used in ML/DL-based applications by other researchers in a wide variety of use cases.
- A set of potential use cases described as possible application scenarios for the benchmark and the published dataset. These use cases are partially solved as a preliminary exploration of the dataset, so that other researchers know how to apply their ML/DL algorithms. The code used in these use cases is also publicly available at [11]. Besides, these use cases are also demonstrated in a real world IoT agriculture deployment using 3 RockPro64 devices, another SBC model.

The remainder of this article is structured as follows. Section 2 provides a review of the literature status regarding SBC performance benchmarking. Section 3 describes the methodology and approach followed for the benchmark implementation and generation of the dataset. Section 4 describes the functions executed in each one of the components considered, while Section 5 explores the collected data through a set of use cases. Section 6 depicts a real-world adaptation and deployment of the benchmark. Section 7 draws the main strong points of the proposed method together with the drawbacks identified during the work development. Finally, Section 8 draws the main conclusions of the present work and future research lines.

2. Related work

This section analyzes related work dealing with performance benchmarking, with a special focus on Edge computing and IoT, and existing datasets regarding IoT device performance monitoring.

Regarding performance benchmarking, Varghese et al. [3] surveyed the evolution of this field from the early 90s to 2020, with special consideration of Edge benchmarking since the 2010s. This survey shows that many performance benchmarking applications have been published in recent years, most of them centered on SBCs, such as Raspberry Pi. The vast majority of these applications focus on CPU and memory benchmarking [15], while only a few test additional resources [24], such as storage, network or accelerators (GPU/TPU). Despite the usefulness to test devices from different brands and models, none of the existing Edge performance benchmarks revised in [3] is focused on the extraction of low-level information capable of detecting hardware imperfections or malfunctioning. In contrast, these existing benchmarks are based on the execution of complex or advanced applications [17], such as AI libraries [9] or orchestrators, and not on fast execution code for low-level fingerprinting. Some

Table 1
Comparison of IoT Benchmarking applications and datasets.

| Solution | Device type | Monitored components | Metrics | Public data | Open source |
|------------------|--------------|----------------------------------|--|--------------------|-------------|
| [13] | SBC | GPU | Processing time, resource usage (%) | ✗ | ✗ |
| [14] | SBC | CPU | GFLOPS, energy | ✗ | ✗ |
| [9] | SBC/PC | CPU, GPU, memory | Processing time, mem speed, energy | ✗ | ✓ |
| [15] | Medical IoT | CPU, memory | Hardware Performance Counters | ✗ | ✓ |
| [16] | Cloud | CPU, memory, network | Processing time, network/mem speed | ✗ | ✗ |
| [17] | SBC | CPU, memory, network | Processing time, network/mem speed | ✗ | ✓ |
| [18] | Edge servers | CPU | N Instructions | ✓(2.5 MB) | ✓ |
| [19] | SBC | CPU | GFLOPS | ✗ | ✓ |
| [20] | IoT | CPU | Packet inter-arrival time | ✓(1.5 GB) | ✗ |
| [21] | IoT | Radio transmitters | Raw transmission data | ✓(+50 GB) | ✗ |
| [22] | SBC | CPU | GFLOPS, energy | ✗ | ✓ |
| [23] | Any | SRAM and Flash memories | Initial bit status | ✓(To be published) | ✓ |
| This work | SBC | CPU, GPU, Memory, Storage | Cycles counters and processing time | ✓(4 GB) | ✓ |

benchmarks [16] also measure the performance of cloud platforms oriented to the IoT, such as AWS Greengrass or Azure IoT Edge. Regarding measurement metrics, most of the benchmarks use time-based metrics such as GFLOPS (Giga Floating Point Operations per Second) for CPU or MB/s for memory and network, with only a few of them using more complex and low-level ones such as Hardware Performance Counters [15]. Besides, although most benchmarking applications provide datasets with them (12 of 14 analyzed applications), only 5 out of 14 benchmarking applications use full open-source software, while 10 out of 15 use commercial-grade software. Pincheira et al. [25] benchmarked six IoT hardware platforms in terms of support to blockchain-based agriculture applications. Finally, [3] also shows that most high-level benchmarks use commercial or proprietary software in their implementations.

In the area of low-level benchmarking, [13] evaluated the performance of the GPUs embedded in ARM SBCs, noticing great improvements comparing the GPU performance to the CPU when doing mathematical operations, but with higher energy consumption. Furthermore, [14] built a Raspberry Pi cluster and performed CPU and energy consumption testing to find the best energy/price/performance model. Similarly, three clusters, each consisting of 16 different SBC models, were built in [22] to benchmark the SBC performance in terms of computing and energy consumption. [19] followed a similar cluster-oriented benchmarking but focusing on cryptography libraries. However, these benchmarks perform fairly simple metrics about performance and do not publish their data, which do not enable the application of the generated data to new domains such as fingerprinting. From a different domain, [26] explored recently the low-level benchmarking of quantum computers, an area gaining importance in recent years that supports the need for lower-level hardware benchmarking.

Dealing with datasets about hardware performance, there are just a few examples available in the literature. Many benchmarking applications include simple data samples [3]. One example is [18], which contains 2.5 MB of traces of different high-level benchmarking applications executed in Edge servers. However, these are not exhaustive datasets collected during long execution periods and are not suitable for ML/DL approaches due to their size constraints. Regarding datasets directly focused on low-level performance fingerprinting, [23] contains fingerprints from different SRAM (Static RAM) chips, which were used in [27] to perform individual identification. However, most SBC models do not include SRAM chips due to their higher cost. From a different perspective, [21] contains radio spectrum measurements from different IoT devices, which can be employed to fingerprint their transmission performance and properties. Similarly, [28] also contains raw IQ signals from 9 IoT devices that can be used for fingerprinting tasks. Moreover, [20] presents inter-arrival time information from different wireless routers and IoT devices, and it is aimed at individual and device type fingerprinting. In contrast, to the best of our knowledge, there is not any comprehensive dataset regarding low-level performance fingerprinting or benchmarking of hardware components.

Table 1 shows a comparison between the different benchmarking applications and datasets found in the literature and the present one. From the analysis made in this section, it is noticed that there is a gap regarding solutions focused on low-level benchmarking. Most of the recent solutions focus on high-level application benchmarking, and the ones focusing on low-level performance only use simple performance metrics and do not provide extensive datasets to enable ML/DL-based use cases or new applications. Moreover, the datasets found in the literature are focused on other areas such as device identification, and not on hardware component benchmarking.

3. Benchmark and dataset generation methodology

This section describes the methodology followed in order to implement the benchmark application and collect the samples available in the dataset: providing the details of the scenario used for data collection; describing the components monitored and how their performance is measured; detailing the libraries used to collect each metric; and finally, explaining the configuration options and measures taken to ensure stability and avoid noise in the samples published.

Table 2

Number, model and most relevant characteristics of the devices used for validation.

| N | RPi model | Revision | SoC | CPU | GPU | Cache | RAM | SD |
|----|-------------------------|----------|---------|--|-------------------------------------|---|------------------|--|
| 15 | Raspberry Pi 4 Model B | 1.1/1.4 | BCM2711 | 1.5 GHz quad-core 64 bit ARM A72 | 500 MHz Broadcom VideoCore VI | 2-way set associative 32 kB and 48 kB level one instruction and data caches, respectively, and a 1 MB unified level two cache | 4 GB LPDDR4 | 16 GB A1 Type 10 SandDisk Ultra |
| 10 | Raspberry Pi 3 Model B+ | 1.3 | BCM2837 | 1.4 GHz quad-core 64 bit ARM A53 | 400 MHz Broadcom VideoCore IV | 2-way set associative 16 kB level one instruction and data caches, and 512 kB unified level two cache | 1 GB LPDDR2 | 16 GB A1 Type 10 SandDisk Ultra |
| 10 | Raspberry Pi Model B+ | 1.2 | BCM2835 | 700 MHz single-core 32 bit ARM 1176JZF-S | 400 MHz Broadcom VideoCore IV | 2-way set associative 16 kB level one instruction and data caches, and 128 kB unified level two cache | 500 MB LPDDR2 | 16 GB A1 Type 10 SandDisk Ultra |
| 10 | Raspberry Pi Zero | 1.3 | BCM2835 | 1 GHz single-core 32 bit ARM 1176JZF-S | 400 MHz Broadcom VideoCore IV | 2-way set associative 16 kB level one instruction and data caches, and 128 kB unified level two cache | 500 MB LPDDR2 | 16 GB A1 Type 10 SandDisk Ultra |

3.1. Deployment and configuration

For the LwHBench benchmark implementation and testing, a wide number of devices is required. In this sense, the benchmark is executed, to collect the dataset, in a set of 45 physical devices composed of several models of Raspberry Pi (RPi) devices. Table 2 shows a summary of the devices employed for validation, their distribution and main characteristics.

All devices of the setup have identical software images, using *Raspbian 10 (buster) 32 bits* as OS and *Linux kernel 5.4.83*. The only variation in the kernel version is related to the core architecture of each device model, *ARMv6* for RPi1/Zero, *ARMv7* for RPi3, and *ARMv8* for RPi4.

Besides, to reduce physical context as much as possible, all devices are located in the same lab room with identical cases and aluminum heat sinks.

3.2. Monitored components

For reliable time/performance measurements of hardware components, the ideal setup is to use as reference a physical oscillator independent of the component being measured (the frequency of the component is not dependent on the reference oscillator). RPi devices include all the main processing components present in a normal computer, including oscillators. However, the number of physical crystal oscillators is reduced to save costs. RPi4 only includes a SoC base oscillator running at 54 MHz and an oscillator for the USB/Ethernet controllers running at 25 MHz. In contrast, RPi3/1/Zero only include one SoC base oscillator running at 19.2 MHz. Then, each component runs at a different frequency using Phase-Locked Loops (PLLs) for base frequency multiplication [29].

This condition implies that some auxiliary components which could be used as reference points, such as the Real-Time Clock (RTC), are simulated. This fact makes it hard to accurately measure the performance and skew of the system from the device itself, as a skew in the CPU timing will affect the time measurements that it is doing of itself.

However, each component can still show performance differences based on the multiplication factor applied to the base crystal oscillator frequency in the associated PLLs. For this reason, each component used to measure the performance of the device is monitored from another component of the device, measuring, in turn, the possible imperfections and deviations between the components. More clearly, for example, the performance of code execution on the CPU is measured in terms of GPU cycles and vice versa.

Following the previous approach, the components whose performance are monitored and stored in the dataset are:

- **CPU.** The execution time of code run in the CPU is measured by monitoring how many GPU cycles have elapsed during that period of time. In this way, the skew between CPU and GPU can be accurately measured. Therefore, the formula to measure CPU performance based on GPU cycle variation is:

$$CPU_{perf.} = \Delta GPU_{cycle_counter} \vdash t_{cpu_code_exec} \quad (1)$$

- **GPU.** In the GPU, the performance of code executed in this component is measured using CPU-based time, just the opposite way to the previous case.

$$GPU_{perf.} = \Delta CPU_{cycle_counter} \vdash t_{gpu_code_exec} \quad (2)$$

- **Memory.** Here, memory read/write operations are also monitored in terms of CPU-based timing, as the RAM chip in RPi has its own functioning frequency different to both the CPU and GPU.

$$Memory_{perf.} = \Delta CPU_{cycle_counter} \vdash t_{mem_ops} \quad (3)$$

- **Storage.** Storage performance measurement is done by input/output operations in the SD card attached to the device with the software image.

$$Storage_{perf.} = \Delta CPU_{cycle_counter} \vdash t_{io_ops} \quad (4)$$

3.3. Benchmark implementation

The LwHBench benchmark code is available in [11]. The data collection program has been implemented using Python 3, so it can be executed as a portable script on any device with the required libraries installed. Besides, this selection has also been influenced by the set of libraries available for CPU and GPU low-level interaction, as it is explained below.

For the GPU low-level interaction and register monitoring, Idein py-videocore [30] and py-videocore6 [31] are employed for VideoCore IV and VideoCore VI GPU-based devices, respectively. To measure GPU cycles, different registers should be considered depending on the GPU version. In particular, the register monitored in VideoCore VI is *CORE_PCTR_CYCLE_COUNT*, while in VideoCore IV the monitored registers are the *performance counters 13-19* [32].

When accessing the CPU cycle counter, there are two different possibilities. The first is to generate a kernel module to enable reading the cycle counter from *userspace*. The second consists of using the interfaces provided by performance monitoring tools such as *perf*. In the code, both approaches are tested.

Regarding the kernel module, the CPU cycle counter is stored in ARMv7 and ARMv8 AArch32 processors using *c15 Cycle Counter Register (CCNT)* and, by default, it can only be read in kernel space. Therefore, a custom kernel module is implemented to allow access to this register (*enable_ccr* folder in [11]). Once compiled, the kernel module should be loaded using *insmod* command with *root* privileges. Finally, the register can be read using the assembly operation *MRC p15, 0, <Rd>, c15, c12, 1*, where *<Rd>* represents the variable where to store the register value. To access *perf* time counting, the easiest method is to use the *time* built-in Python library, which includes *perf_counter_ns()* function to retrieve time in nanoseconds using the previous counter. After some experimentation, it was decided to follow the *perf*-based approach due to its simplicity compared to using a kernel module and assembly code, and its similar consistency in the performance measurements.

Moreover, to automatize the data collection process, a system service has been implemented (*data_collection.service* in the code folder). This service is in charge of the automated data collection script launching and periodic system rebooting in order to reduce the noise introduced by possible factors related to the system running time. Concretely, each device is rebooted after 800 samples are collected.

3.4. Device setup for component stability and isolation

One of the most critical aspects of collecting reliable samples is to ensure that the conditions in the device are as constant as possible, reducing potential sources of noise in the samples. To this end, a number of measures are taken to counteract the impact of other processes running on the device. Specifically, the measures implemented to ensure stability are:

- **Fixed CPU/GPU/RAM frequency.** By default, the kernel dynamically manages the frequency of the device components to save energy when no high task load is present. However, this dynamicity affects to the stability of the performance measurements. Therefore, a fixed frequency is required in the components to measure. In RPi, the frequency of the components can be set to be constant at the maximum using *turbo_mode=1* boot option. Besides, if only a fixed CPU frequency is wanted, *performance* can be used as *scaling_governor* option.
- **Kernel level priority.** Enabling a high priority for the data collection process minimizes the interruptions caused by other programs, removing noise and inaccurate measurements. The best option here is to set the process with the highest scheduling priority. If root privilege is available, using the command *chrt -rr XX* when launching the program enables the “real-time scheduling” of the process, just like a kernel process. If it is not possible to use kernel priority, another option is to use *nice -n -20* to set the maximum *user level* priority.
- **Disable Memory Address Space Layout Randomization (ASLR).** Memory random address organization can affect the stability of memory-related measurements; therefore, this characteristic was disabled during data collection. It can be done using *sysctl kernel.randomize_va_space=0* command, but note that this should be only enabled during memory-related data acquisition, as having ASLR disabled increases the facility to perform memory-based attacks such as buffer overflows.
- **Profiled Guided Optimization (PGO).** PGO is a compiler option intended to improve runtime performance based on static program analysis of code. Python interpreter can be compiled to use PGO by using *-enable-optimizations* option. Furthermore, *Python garbage collector* is also disabled to avoid unintended tasks during the execution.
- **Fixed hash seed.** As hash-based CPU performance measurements are generated, using a fixed seed improves the deterministic characteristics of the function. This is set using *PYTHONHASHSEED=0* (or any other number) as environment variable when running the data collection script. This option should only be used for benchmark, as it can lead to an attacker causing a Denial-of-Service (DoS) by using worst-case performance inputs to the function, which have $O(n^2)$ complexity.
- **Core isolation.** For the multi-core CPU devices, e.g. RPi3 and RPi4, one core is isolated from the rest to execute the benchmark on it using *cpu affinity*. This setup avoids the (kernel) interruptions caused by other processes running in the same CPU core while the data is being generated. Concretely, the kernel options employed were: *i) isolcpus*, to avoid the kernel to schedule any process in that core; *ii) nohz_full*, to tell the kernel to remove as much kernel noise as possible, such as tick interrupts; and *iii) rcu_nocbs*, to offload Read-Copy-Update (RCU) threads and callbacks.

Table 3
Features gathered during data collection.

| Component | Function | Monitored feature |
|-----------|---------------|--|
| – | Timestamp | Unix timestamp |
| | Temperature | Device core temperature |
| CPU | 1 s sleep | GPU cycles elapsed during 1 s CPU sleep |
| | 2 s sleep | GPU cycles elapsed during 2 s CPU sleep |
| | 5 s sleep | GPU cycles elapsed during 5 s CPU sleep |
| | 10 s sleep | GPU cycles elapsed during 10 s CPU sleep |
| | 120 s sleep | GPU cycles elapsed during 120 s CPU sleep |
| | String hash | GPU cycles elapsed during a fixed string hash calculation |
| | Pseudo random | GPU cycles elapsed while generating a software pseudo-random number |
| | Urandom | GPU cycles elapsed while generating 100 MB using <code>/dev/urandom</code> interface |
| GPU | Fib | GPU cycles elapsed while calculating Fibonacci number for 20 using the CPU |
| | Matrix mul | CPU time taken to execute a GPU-based matrix multiplication |
| | Matrix sum | CPU time taken to execute a GPU-based matrix summation |
| | Scopy | CPU time taken to execute a GPU-based graph shadow processing |
| Memory | List creation | CPU time taken to generate a list with 1000 elements |
| | mem reserve | CPU time taken to fill 100 MB in memory |
| | csv read | CPU time taken to read a 500 kB csv file |
| Storage | read ×100 | 100 CPU time measurements for 100 kB storage read operations |
| | write ×100 | 100 CPU time measurements for 100 kB storage write operations |

4. LwHBench benchmark and dataset

This section details the operations executed by the benchmark and the events collected as features for the generation of the dataset associated with this paper. Note that this list can be updated by any other researcher just changing the code of the benchmarking script [11].

As detailed in Section 3, the components leveraged for benchmarking are: CPU, GPU, Memory and Storage. They have been selected because they are the most common hardware elements in any SBC (and generic computer). Table 3 shows the list of features collected for each device component. As it can be appreciated, any of the operations measures typical processing power metrics such as GFLOPS. This is because they are not well suited for low-level component characterization. Besides, they have already been gathered in several previous studies, as shown in Section 2. Concretely, the operations implemented as proof of concept are:

- *CPU*. Different sleep times (from 1 to 120 s) are monitored, trying to measure the accuracy for time keeping in the component. Additionally, some quick-execution functions are monitored: hash calculation of a string, pseudorandom number generation, random number generation using `/dev/urandom` interface, and Fibonacci number calculation.
- *GPU*. Three simple operations are monitored in terms of CPU time: a matrix multiplication, a matrix summation and the processing of a graphic shadow.
- *Memory*. The operations executed are the generation of a list object with 1000 integers, the reserve of 100 MB of data and the time to read a 500 kB csv file. These operations are measured also in terms of CPU time and represent 3 features in the generated data vector.
- *Storage*. 100 read and write operations of 100 kB of data are monitored in the device SD card, generating 200 features in total.

In the dataset [12], each value of Table 3 represents one feature in the vectors (each dataset entry). Besides, each vector ends with the MAC address of the device, which can be used as the label in supervised ML/DL tasks. The data of each device is stored in a csv file, whose name is the MAC address. Additionally, a text file named *MAC-Model.txt* contains the association between the MAC and the model of each device in the testbed.

For data collection purposes, an additional Linux service has been developed (*data_collection.service*). It is in charge of launching LwHBench when the device is booted and it takes care of rebooting the device once 800 samples have been collected. This reboot is done to minimize the possible impact of running time in the collected data (e.g. memory usage of persistent processes). Fig. 1 shows the flow diagram of the data collection performed in each device.

A total of 2 386 126 vectors are available in the dataset, making 4 GB of data. Fig. 2 shows the number of samples per hour from each device model during the data collection period. It can also be seen how some devices went offline during the data collection, corresponding to each of the downward jumps shown in the RPi4 and RPi3 graphs. The dataset contains per device model: 505 584 samples of RPi 1B+, 784 095 samples of RPi4, 547 800 samples of RPi3 and 548 647 samples of RPiZero. With more than two million vectors, the present dataset is the one with the highest number of samples among those found in the IoT benchmarking literature. Besides, Fig. 3 shows the number of samples per device contained in the dataset. The number varies according to the device model, as more powerful ones generate more data in the same time. Besides, some devices suffered power outages during data collection or they were added lately to the set of available devices. Still, on average, more than 50 000 vectors per device are present. It can also be seen that 5 devices have less data due to interruptions during the data collection process.

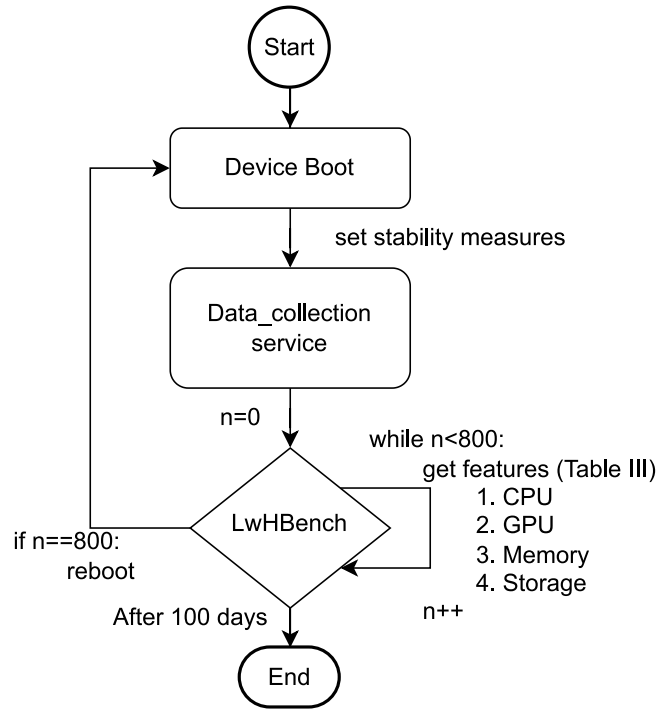


Fig. 1. LwHBench data collection flow diagram.

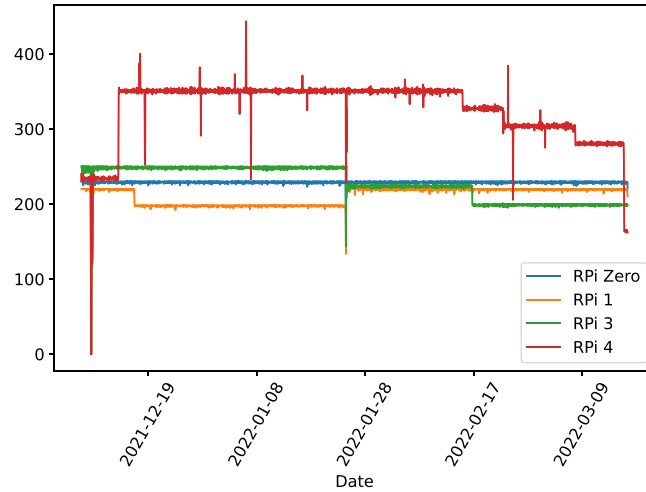


Fig. 2. Samples per hour generated from each device model.

5. Data exploration and use cases

This section explores the dataset described in the previous section. For that purpose, a set of ML/DL-based use cases for network and device management are presented using the data available. Note that the purpose of this section is to show the usefulness of the benchmarking application and the data collected with it, not to find the best solution to the problems proposed as illustrative examples.

5.1. Model/individual identification

The first use case where the dataset (and LwHBench benchmark to generate new data) can be applied is in device identification based on the performance of its device components. This can be a critical task in environments where the devices can be

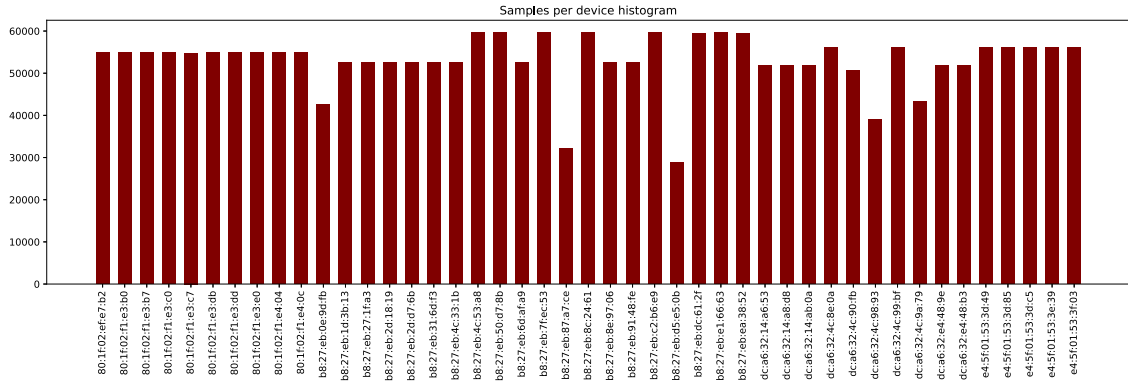


Fig. 3. Samples per device contained in the dataset.

impersonated by malicious ones with new hardware or software configurations [33]. In this sense, this use case can be seen from two different perspectives: *identification of the device model*, a relatively straightforward task from the data measuring the performance of each hardware component, as different hardware will have different performance values; and *individual identification of each device*, a more complex task since to uniquely identify devices with identical software and hardware it is necessary to analyze the differences and imperfections in the chips of each device [34].

5.1.1. SBC model identification

For this first perspective, a ML/DL-based dimensionality reduction and clustering approach [35] is followed to group together the data from each device model. This approach is selected due to its proven efficacy for automated class inference and identification in unlabeled data applied in many research areas, such as network-based IoT device type inference [36], intrusion detection [37] or even biology [38].

As a proof of concept, it is decided to apply clustering to all CPU, GPU, memory and storage features, discarding timestamps and temperature. As clustering algorithms, several options are tested, concretely *PCA*, *t-SNE* and *umap*, reducing the number of dimensions to two. So, the resultant data can be easily plotted for result explainability. Regarding clustering, *k-means* and *DBSCAN* (*Density-Based Spatial Clustering of Applications with Noise*) algorithms are applied over the data once the dimensions have been reduced to two. As the number of device models is previously known, the number of clusters in the algorithm configuration is set to four.

Fig. 4 shows the results when the combination of *PCA* with *k-means* clustering is applied as data processing approach. Different configurations of dimensionality reduction and clustering algorithms gave similar results. It can be seen how four different groups in the data emerge clearly separated, and how the clustering algorithm is able to associate the points successfully. To verify that the clustering is correct, the number of instances assigned to each of them is compared with the number of samples in the dataset belonging to each device model, as also portrayed in Fig. 2. Thus, the results are:

- *Cluster 0*: 784 095 samples, which coincides with the samples of RPi4.
- *Cluster 1*: 505 584 samples, which coincides with the samples of RPi1.
- *Cluster 2*: 547 800 samples, which coincides with the samples of RPi3.
- *Cluster 3*: 548 647 samples, which coincides with the samples of RPiZero.

From the above results, it can be concluded that LwHBench benchmark and dataset can be applied to solve the model identification problem. This approach has performed perfectly, since the number of samples in each cluster matches one of the RPi models deployed in the testbed. Moreover, it can be seen how Cluster 2 (green color) has its values much more concentrated than Clusters 0 and 3 (blue and red), which indicates higher stability in the values of this type of device.

5.1.2. Individual identification

In this case, the objective is to uniquely identify each one of the 45 devices used for dataset generation. Here, an ML/DL-based classification approach is followed due to its demonstrated performance in IoT device identification tasks [39,40].

For this task, all the features available in the dataset regarding components are used, and in this case, it is also included the temperature as a feature, since the correlation between this and the performance of each component can be one of the patterns that the ML/DL algorithm could detect when identifying each device individually. Besides, the storage-related features, 100 measurements for read time and 100 for write, are preprocessed, calculating the average, median, minimum and maximum for each feature group. As the number of devices to be identified is fixed, the 45 devices used to generate the dataset, ML/DL classification techniques [41] are used to identify each SBC. Therefore, the following techniques are compared: Decision Tree (DT), Random Forest (RF), XGBoost, k-Nearest Neighbors (k-NN), Naive Bayes (NB), Support Vector Machine (SVM) and Multi-Layer Perceptron (MLP). For k-NN, NB, SVM and MLP, normalization is applied using min-max : $x = \frac{x - x_{min}}{x_{max} - x_{min}}$. The dataset is split into

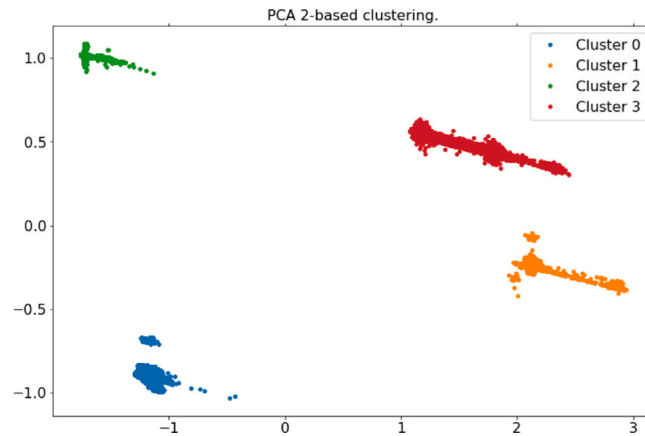


Fig. 4. PCA-based dimensionality reduction with k-means clustering.

Table 4

Device identification results.

| Algorithm | Hyperparameters | Precision | Recall | F1-Score |
|-----------|---|-----------|--------|----------|
| DT | $max_depth = None$ | 0.88 | 0.88 | 0.88 |
| RF | $n_estimators = 100$ | 0.93 | 0.93 | 0.93 |
| XGBoost | $lr = 0.1, gamma = 0.01$ $max_depth = 20$ | 0.97 | 0.97 | 0.97 |
| k-NN | $k = 7$ | 0.32 | 0.32 | 0.31 |
| NB | – | 0.20 | 0.17 | 0.12 |
| SVM | $kernel = linear,$ $gamma = 0.01$ | 0.50 | 0.51 | 0.50 |
| MLP | $1\ relu\ hidden\ layer,$ $50\ neurons$ | 0.58 | 0.57 | 0.56 |

80% of the data for training and cross-validation, and 20% for testing. Table 4 shows the Precision, Recall and F1-Score [42] per algorithm. As it can be seen, XGBoost is the algorithm providing the best results, with a 0.97 in Precision, Recall and F1-Score.

Moreover, Fig. 5 shows the confusion matrix for the 45 devices involved in the identification use case. Note that the RPi model has been added at the beginning of each label to have the devices ordered by model in the image. The results in this use case are satisfactory, as the minimum accuracy in all devices is 0.88, having more than 0.93 in most of them. Therefore, it can be concluded that the collected features are suitable for individual device identification.

Furthermore, more complex approaches could be applied to perform individual device identification depending on the scenario requirements. For example, applying time series approaches or advanced DL models such as LSTM (Long Short-Term Memory) or Transformer networks. One example of these solutions can be found in [43], where similar features to the ones collected about CPU performance were employed to differentiate 25 Raspberry Pi devices. In this case, a sliding window approach is used for vector preprocessing, incorporating new statistical information as features for the ML/DL classifier.

5.2. Performance analysis

The second use case where the benchmark and dataset can be applied is in the comparison of the performance of each device hardware component. This comparison can be seen from two different areas: intra-device comparison, where contextual circumstances such as temperature are analyzed to evaluate their impact on the device performance; and inter-device comparison, where components from different devices, but from the same model and in similar context conditions, are compared to find performance variations based on manufacturing variations.

5.2.1. Intra-device performance analysis

One intra-device use case where the collected dataset can be applied is in the analysis of the impact of temperature variations on the performance of the different device models and their components. This is another research area with a large interest in recent years [44] due to the deployment of SBC in a wide variety of critical scenarios.

For this use case, one of the devices available in the dataset is randomly selected and the impact of temperature on the other hardware-related metrics is analyzed. As the impact of temperature may vary according to the device model, one device per model is selected. Besides, only the first feature regarding storage read and write performance (storage_read_1 and storage_write_1) are

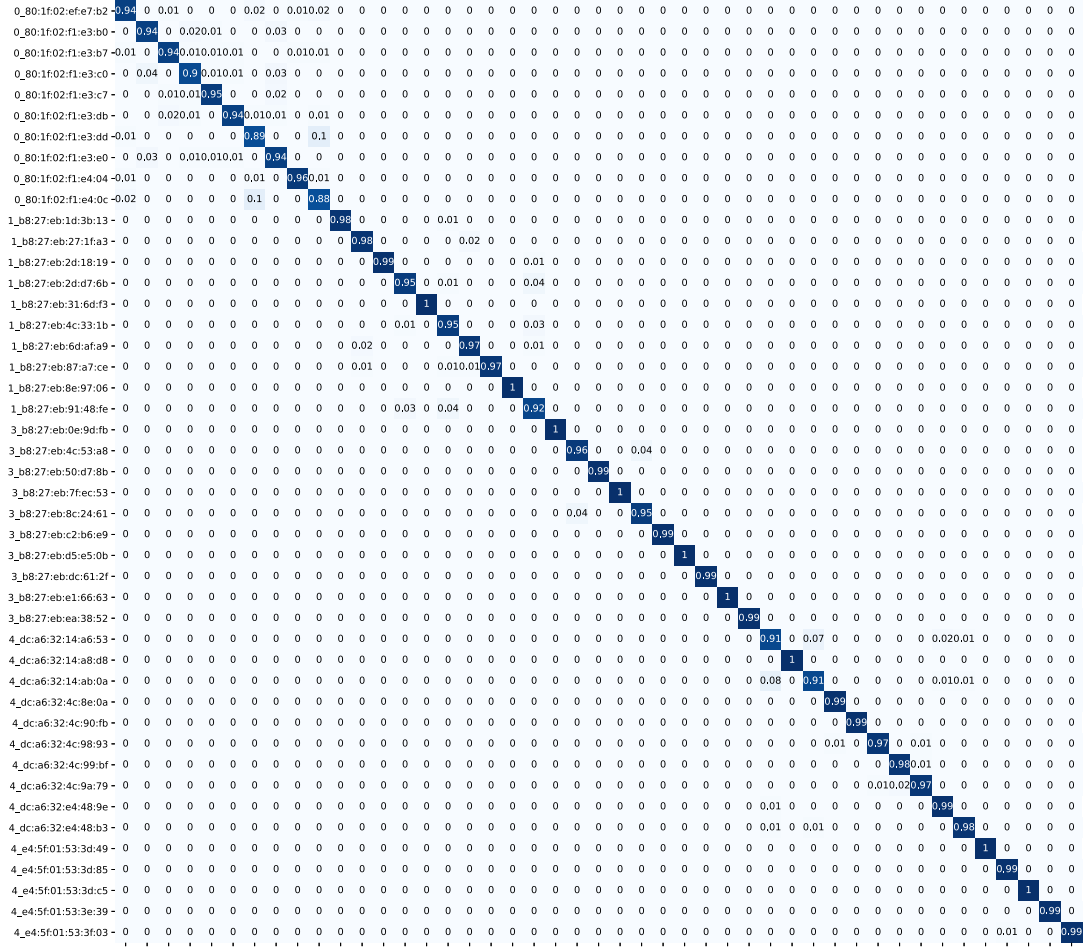


Fig. 5. Individual device identification confusion matrix.

analyzed. In order to test the impact of temperature value on the other features, the correlation between the temperature feature and the other features is studied. The correlation has values between 1 and -1 depending on whether the features increase their value linearly and positively, or inversely and proportionally.

Fig. 6 shows the correlation values for four different devices, one per available model. It can be appreciated how the RPi3 shows a high sensitivity to temperature in almost all features, only the ones based on sleep function execution seems to have a stable performance. This analysis has been repeated with the rest of the RPi3 devices to ensure that it was not a failure in one of them, with all the devices of this model showing very similar correlation graphs. In contrast, the rest of the models show much lower sensitivity to temperature changes, with values close to zero that only vary around ± 0.05 in certain cases and devices.

Thus, from this use case it can be concluded that the RPi3 devices used for the generation of the dataset are the model with the highest sensitivity to temperature changes, as the correlation of this feature with the rest is high in some cases. This fact is interesting for deployments where the physical environment conditions are changing, but the performance is expected to remain stable over time.

Another interesting use case of intra-device analysis is the analysis of how the performance of a device can drop over time due to component wear and tear. For this use case, data needs to be collected over a long period of time. In the case of the available dataset, a total of 100 days (from 6th December 2021 to 17th March 2022) of data have been collected, so although it is not an extensive period, it could give clues about component and device aging.

5.2.2. Inter-device performance analysis

The last use case to be explored is the analysis of the performance variations of different hardware components within devices of the same model. This use case is closely related to individual identification, since it is the performance variations between devices of the same model that can be used to characterize each device separately.

To analyze this use case, the distribution densities of different features for devices of the same model are plotted. As a proof of concept, the selected model is the RPi4 and one feature from each component is shown: *cpu_sleep_120s* for the CPU, *gpu_matrixmul*

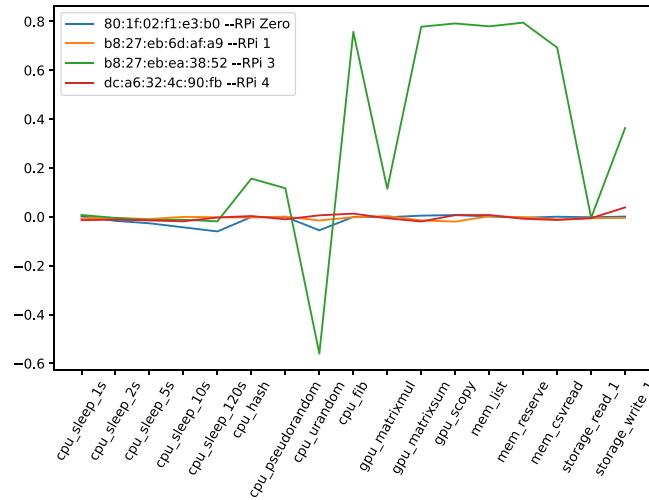


Fig. 6. Temperature correlation plot.

for the GPU, *storage_read_1* for the memory/storage, and *storage_write_1* for the storage. Although identical experiments could be performed for the other models, these are omitted for document space reasons.

Fig. 7 shows the density plots for each feature. Some interesting patterns can be appreciated in these plots. First, it can be seen how for the features *gpu_matrixmul* (Fig. 7(b)) and *storage_read_1* (Fig. 7(c)) the distributions are quite stable between the devices and all of them show similar shapes. However, *cpu_sleep_120s* (Fig. 7(a)) exhibits how each device has a Gaussian distribution centered in a different value, demonstrating that some performance variations are present within the device model. A similar situation is found with *storage_write_1* (Fig. 7(d)) feature, but in this case, the distribution plots tend to have two distinctive center values, and each device sticks to one of them (except for the dark green one with MAC *dc:a6:32:14:a8:d8* that has a higher value, although the distribution shape is similar to the rest).

From this use case, it can be concluded that performance variations are present within each device model depending on the exact device. So, it has been shown that although hardware specifications may be identical, chips contain variations that can be leveraged to perform fingerprinting or identification tasks.

6. Real-world deployment in an agriculture scenario

This section shows the benchmark adaptation and deployment into a real-world IoT environment based on SBC devices. Here, the aim is to show the use case applicability of the proposed solution as well as its adaptability process to other SBC models.

In this sense, an IoT sensor network for agriculture has been considered, being the sensors controlled by three PINE64 RockPro64 devices. As these devices have relatively powerful processing power, they perform the critical task of collecting the sensor data and processing it to infer information about the environment status. Then, they can activate different controllers in order to perform certain tasks, such as watering the plants or rise humidity/temperature, among others.

As the RockPro64 devices maintain control of the farming environment, they are a clear target for a potential attacker looking to extract critical process information or interfere with industrial production. One possible attack could be to replace one of the legitimate devices with an identical one but with modified software to interfere with the normal activity of the environment, for example by rising the temperature and killing the production plants. Therefore, the administrator wants to keep control of the environment, monitoring the performance of the devices during the run time while keeping them identified based on their manufacturing variations. In this sense, this approach can be seen as a real-world application of the individual device identification use case described in Section 5.1.

Each RockPro64 device includes a 6-core CPU: $4 \times$ ARM Cortex A53 cores @ 1.4 GHz + $2 \times$ ARM Cortex A72 cores @ 1.8 GHz; as GPU it features ARM Mali T860; and 2 GB LPDDR4 RAM. As operating system, they use 64 bit armbian (Debian-based Linux for ARM). As the GPU is different from the ones included in RPi devices, the code needed to be adapted to gather the counters from the ARM Mali T860 GPU. Concretely, the *GPU_ACTIVE* counter was selected from the ones available [45]. For cycle counter collection, *ARM HWCPipe* library [46] has been employed. For the CPU-based time gathering, *perf* time is gathered in the same way that for RPi devices, using the *perf_counter_ns()* function. The functions executed are the same as the ones depicted in Table 3, adapted in the case of the GPU to the new hardware using the *ARM Compute Library* [47]. The code is also available in [11].

For experimentation purposes, the code was deployed on the three identical devices for one week. ≈ 35.9 MB of data were collected, with a total of ≈ 12800 vector samples (around 4000 per device). After the data were gathered, the approaches shown in Section 5.1 were implemented to monitor the performance differences between devices and perform individual identification. Fig. 8 shows the distribution plot for the *CPU_sleep_120s* feature. As it can be seen, the three device distributions are clearly differentiated

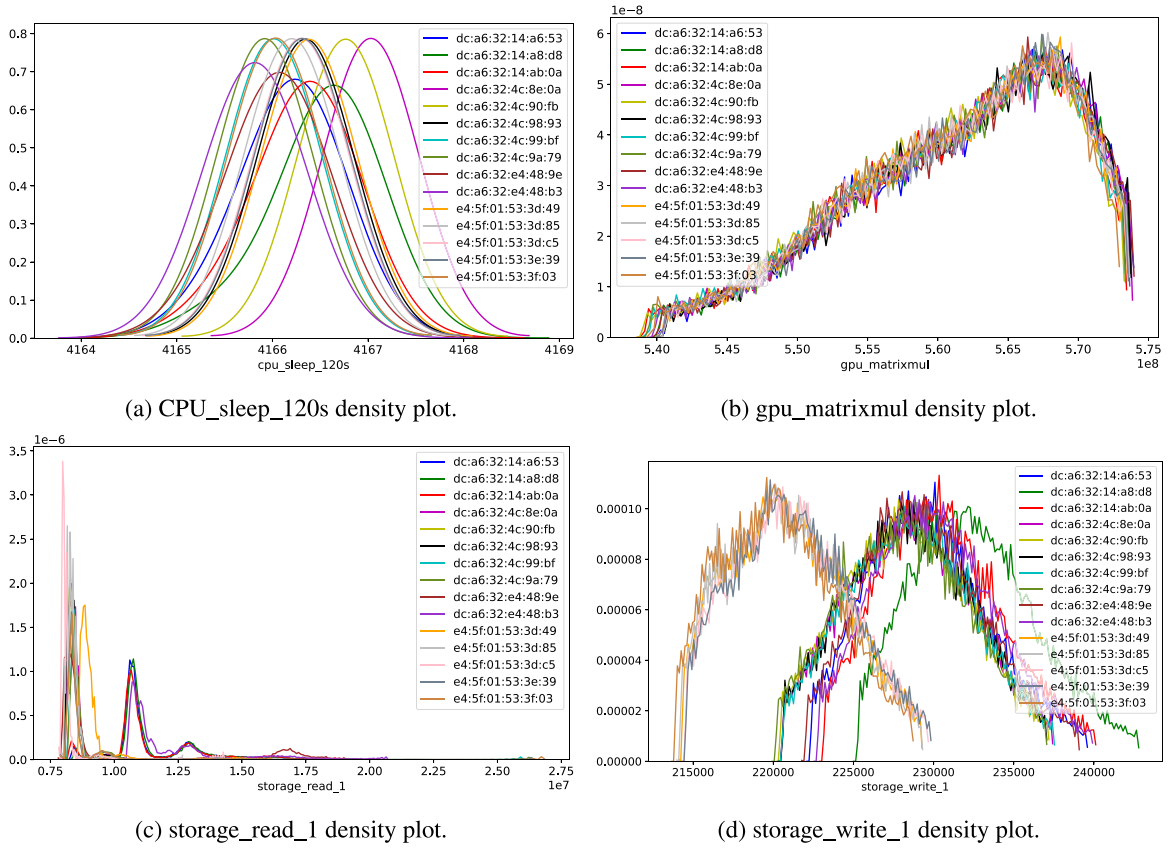


Fig. 7. Raspberry Pi 4 feature density plots.

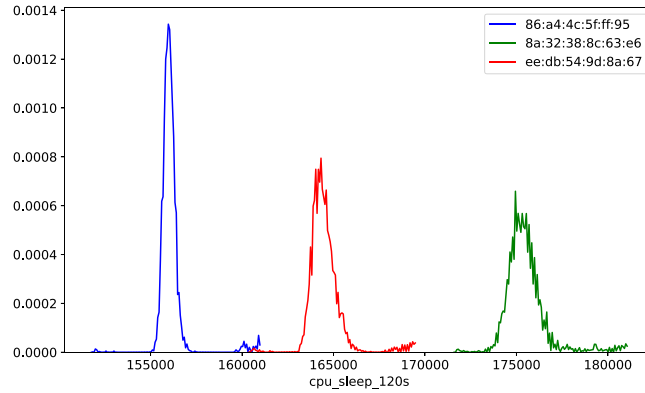


Fig. 8. RockPro64 SBC CPU_sleep_120s density plot.

between each other. This performance variation allows the individual recognition of the devices, as their performance distribution does not overlap. Therefore, the classification results, using the same ML classification algorithms as the ones depicted in Table 4, are perfect, giving 100% F1-Score and accuracy.

With this configuration, the benchmarking application is capable of verifying that the devices deployed in the environment are the legitimate ones. This real-world deployment demonstrates how the benchmarking application can be adapted and deployed in new SBC models with a relatively low effort, enabling the performance analysis of the devices where the application is deployed. This performance analysis enables different use cases, such as individual identification, to be applied in the scenario. However, some challenges were noticed during this real-world deployment. One of the main ones is the necessity of drivers to be able to interact with the CPU/GPU components in order to extract the cycle counters. This is a drawback due to the large amount of hardware

component versions available and the lack of proper documentation in some cases. The positive and negative outcomes of the complete benchmark implementation and data collection are described in the following section, Section 7.

7. Discussion

This section seeks to analyze the main advantages and weaknesses of the proposed benchmarking application and the associated dataset, highlighting the main lessons learned during the work development. From the advantages point of view, it is worth noting the next aspects:

- **An important literature gap has been covered.** As Section 2 shows, there is no low-level benchmarking application for SBC devices, enabling precise hardware analysis, nor any dataset regarding low-level SBC performance. The present work has partially covered these issues with the implementation of the benchmarking application and the release of a large dataset collected for 100 days.
- **Demonstrated utility for ML/DL-based use cases.** The collected dataset has been validated in a set of realistic use cases related to AI-based service management, mainly regarding device identification. Thus, it has been shown the utility of the benchmark when it comes to maintain under control an IoT environment where device fingerprinting is critical.
- **The benchmark is easy to deploy and extend in other RPi-based environments.** The LwHBench benchmark code is completely open-source [11], so other researchers and system administrators can execute it in their own scenarios just by installing the required dependencies. This fact also allows for the extension of the benchmark with new metrics according to the requirements of other scenarios or the available hardware.

Besides, from the drawback perspective, the next points have importance, mainly in future related research:

- **Hardware-based implementation.** LwHBench leverages CPU and GPU cycle counters, which accessed to monitor the performance of other components. Therefore, the implementation for new SBC models may require deep hardware study and understanding. For example, to access to the GPU cycle counter in a new GPU model, it would be needed to read the documentation and try the specific hardware drivers. Besides, due to the large variety in the hardware component versions, some specific components might not have open source drivers or libraries to interact with the performance counters.
- **The collected metrics are not directly applicable for traditional benchmarking.** CPU performance is measured in terms of GPU cycles, an unusual metric that is not very representative of the actual hardware performance when executing high-level tasks. Moreover, the differences between devices from the same model can come both from the measured or the measuring device, so it is difficult to measure where the chip imperfections are actually located.
- **Component isolation might reduce device performance.** The dataset has been collected while executing LwHBench in an isolated core, when possible (RPi3 and RPi4), and reducing the kernel interruptions from other processes to the maximum. Although the other cores can be used without performance limitations, isolating one core can downgrade the performance of critical tasks running in the SBC at the same time. Therefore, further experimentation with weaker isolation measurements can be interesting to know the impact of other processes on the collected values.

This work proposes the only low-level benchmark available in recent literature, and its usefulness has been validated through a series of ML/DL-enabled use cases, which are exciting topics for future IoT-based network and service management solutions. However, as this section details, the proposed solution has room for improvement regarding adaptation to other SBC models.

8. Conclusions and future work

This work has presented a low-level hardware component benchmarking application for SBC, namely LwHBench. It measures the performance of the CPU, GPU, Memory and Storage of the devices using other self-contained components. This approach ensures that the metrics collected are reliable and non-dependent on the imperfections of the component being measured. The benchmark has been implemented for Raspberry Pi devices, for all the models currently available in the market, from RPiZero to RPi4. In order to ensure optimal measurement stability, every possible action that can help to reduce the noise introduced by other programs running on the device has also been considered, such as isolating the core where the benchmark is running or blocking kernel interrupts.

In addition to the application implementation, an exhaustive dataset has been collected from running LwHBench benchmark on a set of 45 devices over 100 days, which contains more than 2 million vectors and 4 GB of data. Subsequently, to explore the available data and show possible areas of use, a series of use cases have been described and partially solved, reflecting the real needs of an environment whose management integrates modern AI-based solutions. These use cases have been divided into two groups: one on device identification, where it has been shown that using LwHBench it is possible to identify both the model and each device individually; and another on performance analysis, where the possible impact of temperature on hardware performance as well as the variations between devices of the same model have been studied.

As conclusions wrap up, the previous contributions intend to advance state of the art regarding system and device management, as it enables new solutions that, based on low-level hardware benchmarking and ML/DL techniques, improve the control over the devices deployed in modern networking environments, such as 5G-based industries.

As future work, it is planned to adapt the benchmarking application to other SBC models in order to collect data from them and perform more in-depth experiments regarding single-device identification and performance impact of temperature and device

aging. Moreover, the benchmarking application will continue being executed in the current devices, generating more data in order to analyze new use cases such as device and component aging. Finally, as further research line, it is planned to integrate the benchmarking application with federated learning techniques, so the data is processed directly in the device according to the use case, without requiring to take the data outside the SBC device.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Links to the data and code included in the manuscript

Acknowledgments

This work has been partially supported by (a) the Swiss Federal Office for Defense Procurement (armasuisse) with the TREASURE and CyberSpec (CYD-C-2020003) projects and (b) the University of Zürich UZH, Switzerland.

References

- [1] Bjørn Andersen, P.-G. Pettersen, *Benchmarking handbook*, Springer Science & Business Media, 1995.
- [2] Roger W. Hockney, *The Science of Computer Benchmarking*, SIAM, 1996.
- [3] Blessen Varghese, Nan Wang, David Bermbach, Cheol-Ho Hong, Eyal De Lara, Weisong Shi, Christopher Stewart, A survey on edge performance benchmarking, *ACM Comput. Surv.* 54 (3) (2021) 1–33.
- [4] Lizy Kurian John, Lieven Eeckhout, *Performance Evaluation and Benchmarking*, CRC Press, 2018.
- [5] P.M. Sánchez Sánchez, J.M. Jorquera Valero, A. Huertas Celdrán, G. Bovet, M. Gil Pérez, G. Martínez Pérez, A survey on device behavior fingerprinting: Data sources, techniques, application scenarios, and datasets, *IEEE Commun. Surv. Tutor.* 23 (2) (2021) 1048–1077.
- [6] Mahmoud M. El-Halwagi, *Sustainable Design Through Process Integration: Fundamentals and Applications To Industrial Pollution Prevention, Resource Conservation, and Profitability Enhancement*, Butterworth-Heinemann, 2017.
- [7] Christopher Michael Wyant, Christopher Robert Cullinan, Timothy Richard Frattesi, *Computing performance benchmarks among CPU, GPU, and FPGA*, Computing (2012).
- [8] Shi Dong, Ping Wang, Khushnood Abbas, A survey on deep learning and its applications, *Comp. Sci. Rev.* 40 (2021) 100379.
- [9] Xingzhou Zhang, Yifan Wang, Weisong Shi, pCAMP: Performance comparison of machine learning packages on the edges, in: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.
- [10] Stephan Patrick Baller, Anshul Jindal, Mohak Chadha, Michael Gerndt, DeepEdgeBench: Benchmarking deep neural networks on edge devices, in: *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 20–30.
- [11] P.M. Sánchez Sánchez, LwHBench benchmarking application for raspberry pi, 2021, [Online; accessed 8-March-2022] <https://github.com/sxz0/LwHBench>.
- [12] Pedro Miguel Sánchez Sánchez, José María Jorquera Valero, Alberto Huertas Celdrán, Gérôme Bovet, Manuel Gil Pérez, Gregorio Martínez Pérez, LwHBench dataset, 2022, IEEE DataPort.
- [13] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, Alex Ramirez, Energy efficient HPC on embedded SoCs: Optimization techniques for Mali GPU, in: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 123–132.
- [14] Michael F. Cloutier, Chad Paradis, Vincent M. Weaver, A Raspberry Pi cluster instrumented for fine-grained power measurement, *Electronics* 5 (4) (2016) 61.
- [15] Ankur Limaye, Tosiron Adegbiya, HERMIT: A benchmark suite for the internet of medical things, *IEEE Internet Things J.* 5 (5) (2018) 4212–4222.
- [16] Anirban Das, Stacy Patterson, Mike Wittie, EdgeBench: Benchmarking edge computing platforms, in: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018, pp. 175–180.
- [17] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, Blessen Varghese, DeFog: Fog computing benchmarks, in: *4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.
- [18] Klervie Toczé, Norbert Schmitt, Ulf Kargén, Atakan Aral, Ivona Brandić, Edge workload trace gathering and analysis for benchmarking, in: *2022 IEEE 6th International Conference on Fog and Edge Computing, ICFEC*, 2022, pp. 34–41.
- [19] Daniel Hawthorne, Michael Kapralos, Raymond W Blaine, Suzanne J Matthews, Evaluating cryptographic performance of Raspberry Pi clusters, in: *2020 IEEE High Performance Extreme Computing Conference, HPEC*, 2020, pp. 1–9.
- [20] A.S. Uluagac, CRAWDAD dataset gatech/fingerprinting (v. 2014-06-09), 2014, [Online; accessed 31-March-2022] <https://crawdad.org/gatech/fingerprinting/20140609>.
- [21] A.K. Hagelskjær, B.H. Grevenkop-Castenskiöld, M.H. Jespersen, T. Arildsen, E. Carvalho, P. Popovski, IoT device identification dataset, 2020, [Online; accessed 31-March-2022] <https://doi.org/10.5281/zenodo.3638165>.
- [22] Philip J. Basford, Steven J. Johnston, Colin S. Perkins, Tony Garnock-Jones, Fung Po Tso, Dimitrios Pezaros, Robert D. Mullins, Eiko Yoneki, Jeremy Singer, Simon J. Cox, Performance analysis of single board computer clusters, *Future Gener. Comput. Syst.* 102 (2020) 278–291.
- [23] Yang Su, Yansong Gao, Surya Nepal, Damith C. Ranasinghe, NoisFre: Noise-tolerant memory fingerprints from commodity devices for security functions, 2021, IEEE DataPort.
- [24] Flávio Ramalho, Augusto Neto, Virtualization at the network edge: A performance comparison, in: *2016 IEEE 17th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016, pp. 1–6.
- [25] Miguel Pincheira, Massimo Vecchio, Raffaele Gialfreda, Benchmarking constrained IoT devices in blockchain-based agri-food traceability applications, in: *Blockchain and Applications: 3rd International Congress*, Springer, 2022, pp. 212–221.
- [26] K. Wright, K.M. Beck, S. Debnath, J.M. Amini, Y. Nam, N. Grzesiak, J.-S. Chen, N.C. Pistenti, M. Chmielewski, C. Collins, K.M. Hudek, J. Mizrahi, J.D. Wong-Campos, S. Allen, J. Apisdorf, P. Solomon, M. Williams, A.M. Ducore, A. Blinov, S.M. Kreikemeier, V. Chaplin, M. Keesan, C. Monroe, J. Kim, Benchmarking an 11-qubit quantum computer, *Nature Commun.* 10 (1) (2019) 1–6.
- [27] Yansong Gao, Yang Su, Surya Nepal, Damith C. Ranasinghe, NoisFre: Noise-tolerant memory fingerprints from commodity devices for security functions, 2021, arXiv preprint arXiv:2109.02942.

- [28] Gianmarco Baldini, IoT transient radio frequency signals, 2020, IEEE DataPort.
- [29] Embedded Linux Wiki, The undocumented Pi, 2021, [Online; accessed 8-March-2022] https://elinux.org/The_Undocumented_Pi.
- [30] Idein, Py-videocore. Python library for GPGPU on Raspberry Pi, 2021, [Online; accessed 8-March-2022] <https://github.com/nineties/py-videocore/>.
- [31] Idein, Py-videocore6. Python library for GPU programming on Raspberry Pi 4, 2021, [Online; accessed 8-March-2022] <https://github.com/idein/py-videocore6/>.
- [32] Broadcom, VideoCore IV 3D architecture reference guide, 2013, [Online; accessed 30-March-2022] <https://docs.broadcom.com/doc/12358545/>.
- [33] Leonardo Babun, Hidayet Aksu, A. Selcuk Uluagac, CPS device-class identification via behavioral fingerprinting: From theory to practice, *IEEE Trans. Inf. Forensics Secur.* 16 (2021) 2413–2428.
- [34] Dror G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*, Cambridge University Press, 2015.
- [35] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, Arthur Zimek, A general framework for increasing the robustness of PCA-based correlation clustering algorithms, in: *International Conference on Scientific and Statistical Database Management*, 2008, pp. 418–435.
- [36] Arunan Sivanathan, Hassan Habibi Gharakheili, Vijay Sivaraman, Inferring IoT device types from network behavior using unsupervised clustering, in: *2019 IEEE 44th Conference on Local Computer Networks, LCN*, 2019, pp. 230–233.
- [37] Liqun Liu, Bing Xu, Xiaoping Zhang, Xianjun Wu, An intrusion detection method for internet of things based on suppressed fuzzy clustering, *EURASIP J. Wireless Commun. Networking* 2018 (1) (2018) 1–7.
- [38] Daniel Granato, Jânio S Santos, Graziela B Escher, Bruno L Ferreira, Rubén M Maggio, Use of principal component analysis (PCA) and hierarchical cluster analysis (HCA) for multivariate association between bioactive compounds and functional properties in foods: A critical perspective, *Trends Food Sci. Technol.* 72 (2018) 83–90.
- [39] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, Vijay Sivaraman, Classifying IoT devices in smart environments using network traffic characteristics, *IEEE Trans. Mob. Comput.* 18 (8) (2018) 1745–1759.
- [40] Ivan Cvitić, Dragan Peraković, Marko Periša, Brij Gupta, Ensemble machine learning approach for classification of IoT devices in smart home, *Int. J. Mach. Learn. Cybern.* 12 (11) (2021) 3179–3202.
- [41] Aized Amin Soofi, Arshad Awan, Classification techniques in machine learning: applications and issues, *J. Basic Appl. Sci.* 13 (2017) 459–465.
- [42] Margherita Grandini, Enrico Bagli, Giorgio Visani, Metrics for multi-class classification: an overview, 2020, arXiv preprint [arXiv:2008.05756](https://arxiv.org/abs/2008.05756).
- [43] Pedro Miguel Sánchez Sánchez, José María Jorquera Valero, Alberto Huertas Celdrán, Jérôme Bovet, Manuel Gil Pérez, Gregorio Martínez Pérez, Can evil IoT twins be identified? Now yes, a hardware behavioral fingerprinting methodology, 2021, arXiv preprint [arXiv:2106.08209](https://arxiv.org/abs/2106.08209).
- [44] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S Unsal, Adrian Cristal Kestelman, Jingwen Leng, Modern hardware margins: CPUs, GPUs, FPGAs recent system-level studies, in: *2019 IEEE 25th International Symposium on on-Line Testing and Robust System Design, IOLTS*, 2019, pp. 129–134.
- [45] Peter Harris, Mali Midgard family performance counters, 2016, [Online; accessed 30-July-2022] <https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/mali-midgard-family-performance-counters/>.
- [46] A.R.M. Developers, HWC Pipe, 2021, [Online; accessed 12-September-2022] <https://github.com/ARM-software/HWC Pipe>.
- [47] A.R.M. Developers, ARM Compute Library, 2021, [Online; accessed 12-September-2022] <https://arm-software.github.io/ComputeLibrary/latest/index.xhtml>.