IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 27, 2020, accepted July 31, 2020, date of publication August 5, 2020, date of current version August 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3014533

Rancid: *Relia*ble Be*nc*hmarking on Andro*id* Platforms

SERGIO AFONSO AND FRANCISCO ALMEIDA
Department of Computer Engineering and Systems, Universidad de La Laguna, 38200 San Cristóbal de La Laguna, Spain

Corresponding author: Sergio Afonso (safonsof@ull.es)

This work was supported in part by the Spanish Ministry of Science, Innovation and Universities under Project PID2019-107228RB-I00 and Contract FPU16/00942, in part by the European Commission (European Regional Development Fund) and the Spanish Ministry of Economy and Digital Transformation (MINECO) under Project TIN2016-78919-R, in part by the Government of the Canary Islands under Project ProID2017010130, and in part by the CAPAP-H Network (Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas).

ABSTRACT Benchmarking is an important step in the code optimization process that enables empirical performance evaluations in computer systems. Application profiling allows the detection of bottlenecks within the code, and benchmarking can be used to measure the effect of optimizations on performance and to compare implementations. However, obtaining reliable and reproducible performance metrics on modern mobile platforms is a complex task that is often overlooked. This is necessary to produce scientifically sound experiments. There are several factors that introduce noise on performance measurements. We identify and measure the most relevant set of these factors and design a methodology that enables more reliable performance benchmarking on Android platforms. We also describe our flexible benchmarking framework, *Rancid*, designed to transparently solve these problems. It enables application developers to quickly obtain reliable performance metrics for their code on a wide set of platforms. The evaluation of our methodology and framework shows an improvement on the behavior of results, successfully providing more precise and reproducible measurements on a range of devices and implementations.

INDEX TERMS Heterogeneous systems, mobile computing, performance of systems.

I. INTRODUCTION

The high rate of development of mobile platforms in the last decade has resulted in increasingly low-power and feature-rich devices with high levels of computational performance. Advances in System on Chip (SoC) technologies, which allow integrating multiple multicore CPUs and accelerators interconnected via a shared memory system, provide the foundation for these developments. The performance capabilities they provide open new application possibilities for mobile platforms, on both smartphones and single-board computers. Increasingly complex computer vision, image processing and artificial intelligence applications can be developed for real time use on these devices, meaning that high-performance code for low-power heterogeneous architectures must be developed and optimized.

Application optimization, like most fields within computer science and science in general, is predominantly an experimental field. Any technique intended to improve any metric of performance has to be eventually evaluated in real systems in order to prove its fitness. As a consequence, it is of paramount importance that measurements are obtained under controlled experiment conditions, such that results can be reproduced reliably and relevant conclusions can be extracted. Otherwise, the observed behavior could be caused by external factors outside of the scope of the experiment. The problem of reproducibility of experimental research in computer science is well known [1], and it is agreed that improvements in evaluation methodologies, along with other factors, must be made to solve it [2].

In the applications optimization field experimentation consists of locating bottlenecks within an application and then measuring the effect of optimizations on these bottlenecks to discover what can reduce their effect and to what extent. Those are, respectively, the stages of profiling and benchmarking. Profiling does not usually have to be as exact as benchmarking, as long as it points to the actual bottlenecks of an application, and developer-friendly tools to that end are already available for most platforms and main programming languages [3]–[6].

Benchmarking, on the other hand, is what allows multiple implementations of an algorithm to be compared against

The associate editor coordinating the review of this manuscript and approving it for publication was Longxiang Gao.

each other and measured with a much higher precision. It is the basis for evaluating hardware acceleration methods and low-level optimizations, as well as the effect of compiler optimizations. Hence, it demands a higher level of rigorousness than profiling. In its simplest form, it can consist of repeatedly running a fixed number of times a piece of code and then finding the average of the metric under observation. However, the measurements that it will provide can be misleading. A much more careful planning and benchmark design is required in order to obtain accurate measurements [7], [8].

Current mobile SoCs present a set of unique characteristics that difficult the benchmarking process. Some of those are related to thermal management, Dynamic Voltage and Frequency Scaling (DVFS) and power limitations [9], and some have been identified before as a problem to performance evaluation [10], but reliable solutions to these problems are still to be found. On the software side, mobile operating systems introduce problems to the benchmarking process as well, due to their interactive, always connected and restricted nature. We focus on Android as it is currently the most widespread mobile operating system. Though applications for Android are mostly written in the Java programming language, previous works about reliable benchmarking for Java [11] cannot be directly applied due to runtime differences on various Android versions that must be considered, as well as platform configuration limitations.

The main features of the measurements that need to be improved are their accuracy, precision and behavior. With accuracy, we refer to how much the magnitude of a set of measurements is centered around its real value within the system that is being measured. The first problem that rises in mobile devices is that we need to define what this real value is. In the case of a modern mobile processor, dynamic changes in clock frequency create different performance levels, so a single performance level must be ensured so that all measurements are done within the same specification.

The precision is the degree of closeness of multiple measurements among each other, and it can be improved by reducing random noise sources within the system. When we speak about the behavior of the measurements, we are talking about the shape of their distribution according to their values. If only random noise is present, measurements should follow a normal distribution, which is usually assumed but not always verified. However, we have found that mobile platforms tend to behave differently due to systematic error sources that impact all measurements differently. Processor frequency changes can result in multi-modal distributions, and temperature can have an impact that gets more pronounced as more measurements are taken. Fig. 1 shows execution time histograms for independent executions of the same benchmark using the same parameters in the same device. It is apparent that behavior will vary widely if no actions are taken to prevent that from happening.

In this work, we focus on the set of unique characteristics of mobile devices that present difficulties to the benchmarking process, and we propose solutions that minimize their impact. We design a benchmarking methodology that ensures the reproducibility of benchmark measurements on mobile platforms, by identifying and tackling several systematic error sources present in these platforms, while significantly increasing the precision of the measurements. Additionally, we build a framework that we use to implement and evaluate our methodology, while being able to be extended to alternative methodologies and easily applied to any applications to evaluate. In summary, we i) provide an automated system to significantly increase the reliability of performance evaluations on mobile platforms; and ii) create the necessary infrastructure to allow other methodologies to be quickly implemented and compared. This differs from existing Java benchmarking frameworks [12], [13] in that it focuses on setting up the system prior to execution, guaranteeing that the execution environment will be stable, while providing tools to change this behavior depending on device features.

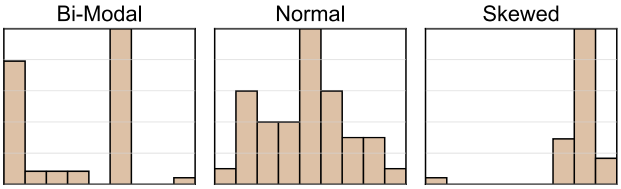


FIGURE 1. Execution time histograms for separate repetitions of the same benchmark. Horizontal axes represent the same range of values on all three graphs. 50 measurements taken per benchmark.

We believe that these are key steps towards advancing the current state of the practice with regards to rigorous empirical evaluations on these types of dynamic computing systems. Our contributions are:

- We build on previous works focused on reliable Java benchmarking in order to tackle major differences in Android's Java execution model compared to state-of-the-art desktop and server virtual machines.
- We identify a range of systematic error sources on mobile platforms that must be mitigated in order to obtain accurate, precise and well-behaved measurements from benchmarks in these environments.
- We design countermeasures against the identified error sources that help reduce their impact. We also provide indirect methods to reduce error for cases when the preferred solution is not possible due to common limitations of hand-held devices. These serve as a set of practical guidelines that help developers improve the reliability of their benchmarks.
- We design and build the *Rancid* framework, a modular and extensible benchmarking framework that isolates the benchmarking methodology from the code to benchmark, so developers can benefit from more reliable results with very reduced additional work and implement and compare alternative benchmarking methodologies.

- We implement our proposed methodology in our benchmarking framework, which can be easily extended to collect any metric the user deems relevant, contains tools for results processing which can be extended as well, and it is compatible with any platform other than Android. It can be used to benchmark code in any system with Java support.
- We evaluate the effectiveness of our reliable benchmarking methodology and framework, comparing our results to baseline values gathered in a straightforward way, on a range of representative devices based on mobile SoCs and CPU implementations on Java and C/C++, and GPU implementations on OpenCL. Multiple independent benchmark executions are used in order to evaluate the reproducibility of the results.

Our experimentation on multiple representative devices, kernels and programming languages shows that our methodology greatly improves the precision, reproducibility and reliability of benchmarks, on both longer and shorter runs, validating the relevance of this work. Although there is still room for improvements, we provide the infrastructure where these can be tested and transparently provided to application developers.

This paper is structured as follows: on Sect. II, we present the runtime features of Android applications and our testing platforms; Section III identifies the set of main systematic error sources and measures each of their impacts on benchmarking in isolation; we propose and evaluate separately countermeasures to minimize each of these errors in Sect. IV; in Sect. V we present a benchmarking framework that integrates many of the proposed countermeasures; an evaluation of the proposed methodology and framework is presented in Sect. VI; and Sect. VII finishes with conclusions.

II. ANDROID RUNTIME FEATURES

A. COMMON RUNTIME FEATURES

Android is currently the most widespread operating system for mobile devices. It is based on a Linux kernel, on top of which a large software stack has been built. This layer provides application developers with abstraction from the hardware and a unified interface for accessing the different supported capabilities provided by the underlying SoC. On top of that, unrestricted or ''root'' access to the system is generally unavailable, which helps with security issues.

Being based on Linux, raw sensor data can be found in mostly vendor-dependent paths in the file system through sysfs [14]. This is sometimes needed when the sensor to be queried is not part of the standard set of sensors that the Android API defines, as is the case with processor frequencies and internal components temperatures. These two metrics are highly correlated to performance, as we show in Sect. III, so monitoring them is a very important step towards reliable benchmarking on mobile platforms.

Processor frequencies in Linux are controlled through what are called CPU/GPU governors. These are heuristics integrated in the kernel that continuously monitor the state of the system and set processor frequencies [15] according to it. Their main goal is usually to provide high performance when necessary, and to optimize energy usage through frequency scaling when possible. Some governors, however, focus on achieving maximum performance or power saving. By default, mobile devices use governors designed for a balance between interactivity, performance and energy consumption, but this means that performance varies over time in a generally unpredictable manner. If it happens while benchmarking, this situation implies a change on the conditions of the experiment that triggers unreliable results.

There is a process, commonly referred to as ''rooting'', which consists in obtaining unrestricted access to the underlying Linux operating system by unlocking the device's bootloader and replacing the system image by one in which privilege scalation is possible. Device manufacturers can decide not to support bootloader unlocking for security and stability reasons, in which case the only way to achieve it would be through a security vulnerability exploit. In order to be able to change system parameters, like frequency governors, root access is necessary. Although certain users go through the rooting process, it is far from common practice and the ability of tuning system parameters cannot be assumed in most cases.

Applications for Android are mainly written in the Java programming language. The majority of tools for Android development are available for Java code, such as a profiler and debugger, and most of the Android APIs are written for Java as well. Although benchmarking of Java applications is already relatively well understood due to previous works, there is a major difference in the way Android handles Java code execution that makes previous knowledge incomplete.

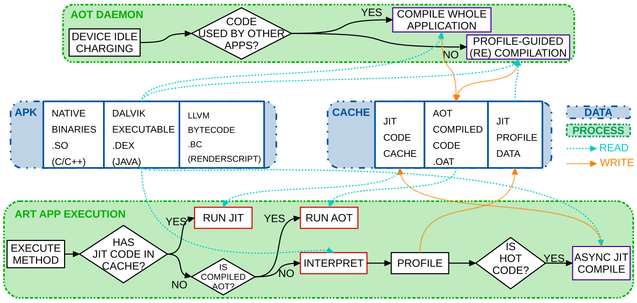


FIGURE 2. Android 7.0 Java execution model.

From Android 5.0 to 7.0, Dalvik bytecode contained in Android Application Packages (APK) is compiled to native binaries supported by the Android Runtime (ART) at installation time in the device [16]. Starting in Android 7.0, ART uses a combination of ahead-of-time (AOT) and just-in-time (JIT) compilation, shown in Fig. 2. Most frequently executed code is AOT compiled in the background when the device is idle, and any code that is being interpreted frequently gets JIT compiled while the application runs. This contrasts with the way regular Java bytecode executes on any other platform, as well as Android before version 5.0, where it is

TABLE 1. Hardware platforms.

	Sony Xperia Z (*SXZ*)	Odroid XU3 (*XU3*)	Huawei P8 Lite (*P8L*)
OS	Android 5.1.1	Android 4.4.4	Android 8.0
Root	No	Yes	Yes
Apps	Regular	Barebones	Barebones
SoC	Qualcomm Snapdragon APQ8064	Samsung Exynos 5422	HiSilicon Kirin 655
CPU	Qualcomm Krait (4-core @ 1.5 GHz)	Arm Cortex-A15 (4-core @ 2.1 GHz) + Arm Cortex-A7 (4-core @ 1.5 GHz)	Arm Cortex-A53 (4-core @ 2.1 GHz) + Arm Cortex-A53 (4-core @ 1.7 GHz)
GPU	Qualcomm Adreno 320 (4 CU @ 400 MHz)	Arm Mali-T628 MP6 (2+4 CU @ 600 MHz)	Arm Mali-T830 MP2 (2 CU @ 900 MHz)
RAM	2GB LPDDR2 @ 533 MHz	2GB LPDDR3 @ 933 MHz	3GB LPDDR3 @ 933 MHz
Cooling	Passive	Active	Passive
Release	2013	2014	2017

interpreted by a Java Virtual Machine and/or JIT compiled. This means that JIT-compilation of hot spots in the code only happens in certain versions of Android, although other features of managed programming languages such as garbage collection overhead still must be considered when designing benchmarks.

In addition to Java, Android supports development on C/C++ and Renderscript as well. Native C/C++ code tends to be useful in order to integrate an application mainly written in Java with native libraries and other available system libraries. This integration is achieved through the Java Native Interface (JNI), which provides a way to implement Java methods in C/C++ and exchange data between the Java managed and C/C++ unmanaged memory spaces. There is a certain overhead involved in passing data between the two memory spaces that needs to be taken into consideration when benchmarking, so that it is only measured if that is the intent of the benchmark.

An important note to make is that the vast majority of mobile platforms nowadays are based on SoCs that integrate GPU and possibly specialized Digital Signal Processors (DSP) along with a multicore CPU. These are all programmed in various ways through software interfaces such as OpenCL, Renderscript and vendor-specific libraries, although there are frameworks that try to reduce the development complexity by unifying many of these interfaces [17], [18]. Writing high performance code for Android often relies on accelerators [19], and that is a case in which benchmarking is of paramount importance, so the interaction between these interfaces must be considered.

B. HARDWARE PLATFORMS

In order to discover and measure the sources of error present when benchmarking Android applications, as well as evaluating the effectiveness of countermeasures designed to mitigate these errors, we use a heterogeneous set of devices. This allows us to represent a large amount of cases that could arise when benchmarking a piece of code in an Android device. A summary of the features of each one of the selected devices is presented in Table 1.

Every device runs a different version of Android, each implementing one of the main Java execution models found in this OS, described in Sect. II-A. Two of the devices have been rooted and stripped from most non-critical applications, whereas the other one is representative of a device used regularly by a user, with no root access and with plenty of applications installed. Each device is based on a SoC from one of the main manufacturers, although they all share Arm or Arm-based CPU and GPU designs, as they are prevalent in this market. One of the considered SoCs contains a single quad-core CPU, whereas the others are based on Arm's dual CPU big.LITTLE architecture.

Lastly, two of the devices are smartphones, whereas the other one is a board that has been built with the same type of processing components. This board contains a fan that is able to actively cool the processor package, in contrast to smartphones that can only reduce temperature via passive means such as reducing processor clock frequencies, switching cores off or reducing processing load.

III. BENCHMARKING ERROR SOURCES

In general, all empirical measurements are subject to noise caused by external factors. In the case of benchmarking, these errors are ubiquitous, so they should always be properly addressed in order to obtain correct and precise measurements. It is common, during the application optimization process, to benchmark multiple implementations of an algorithm in order to select the best performing according to some metric. If enough care is not taken during the design of the benchmark or the interpretation of its results, random factors could skew the comparison and, ultimately, the decision taken.

Random noise has the property of reducing the precision and, hence, the confidence with which evaluators can reliably extract properties or compare the benchmarked code. Its impact can be greatly reduced through repetition and statistical methods. In fact, the benchmarking process essentially intends to estimate the value of a random variable with a high level of confidence. In this case, the variable could be some metric of performance or energy, and its random component would mainly be influenced by sources of random noise present in any computing system. Some of these noise sources are OS process scheduling, memory latency, or CPU branch predictors and pipeline stalls.

On the other hand, there are also systematic error sources that impact all measurements in a similar way, leading to skewed results. These are harder to detect, as they cannot usually be spotted through metrics alone. Some knowledge of the system or expected behavior is required as well. Systematic error sources must be addressed if any proper conclusion is to be extracted from a benchmark.

Some well-known sources of systematic error during application benchmarking are compiler optimization levels or debugging support. In the case of mobile applications and, in particular, Android, there are additional sources of systematic error that should be neutralized. These are usually not a problem or easily solved in most other systems, so their impact on mobile platforms has been neglected by many authors. Our goal is to identify these sources of systematic error, illustrate their impact and provide solutions, defining a set of practical measures that enable developers to obtain deeper and reliable insights from their benchmarks.

It tends to be assumed that multiple repetitions of a benchmark should behave the same way, and large enough samples will fit a normal distribution. This is true of an experiment where system conditions do not change. However, specific measures must be taken to enforce this requirement, given the very dynamic nature of mobile systems' performance. Otherwise, behaviors like the ones presented in Fig. 1, based on data from our experimentation in Sect. VI, will potentially happen, making metrics like the mean or the standard deviation not accurately represent this behavior.

We present a set of benchmarks that show the execution time of a simple Gaussian Blur kernel when repeated multiple times[1] using the same input parameters. Ideally, the execution time should be independent from the moment at which it was obtained, but random and systematic error sources show that this is not the case. We use Java, native (C/C++) and OpenCL implementations in order to evaluate the distinct features of each of those programming models in Android. These benchmarks illustrate the separate effect of every systematic error source we identified, while simultaneously mitigating the rest. We exclude additional warm-up runs for stability from the plotted data points.

Although the impact we measure for each error source independently tends to seem marginal in many cases, we need to consider that it can be more pronounced on other devices and applications. Furthermore, all sources of error are interdependent, compounding their impact into behaviors that can lead to unreliable or inconsistent measurements, as illustrated in Fig. 1.

A. PROCESSOR FREQUENCY

One of the main factors to consider when benchmarking a mobile application is processor frequency. As we discussed in Sect. II-A, CPU and GPU frequencies in Linux are controlled by governors. On hand-held devices, due to their stricter power constraints, we find default governors are more aggressive at lowering frequency than their desktop counterparts. Fig. 3 shows how using the *performance* CPU governor reduces execution time and variability.

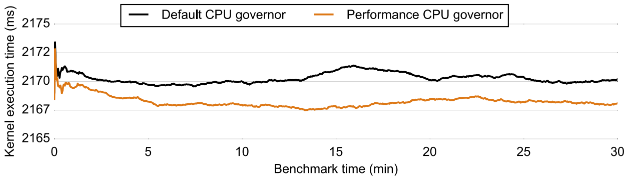


FIGURE 3. CPU governors performance on P8L (C/C++, window[2]).

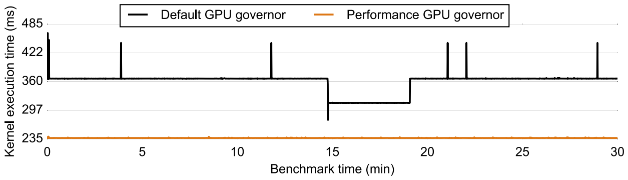


FIGURE 4. GPU governors performance on XU3 (OpenCL).

The governor change is only possible because we have root access to the device, and because the kernel the device is running included a *performance* governor that sets the frequency at its maximum value. In this case the relative improvement is minimal, but its impact depends on the default CPU governor behavior, device and measured code. In the case of GPU governors, we see a much larger relative change of 33.91% in average between variable GPU frequency and maximum performance in Fig. 4. We measured a correlation of 97.51% between GPU frequency and execution time in this case, which could be hinting at a compute-bound benchmark. In any case, we see that fixing processor frequency leads to more stable results, and ignoring its impact can significantly skew them.

B. TEMPERATURE

Although we have observed that frequency changes are detrimental to benchmarking, fixing processor frequency makes other sources of error more pronounced. One of the main reasons, other than power consumption, why mobile devices reduce processor and memory frequency is temperature. Since the vast majority of these devices are passively cooled, when temperatures rise above a certain threshold, the frequency is reduced in order to reduce heat output, avoiding user discomfort and protecting internal components. This effect is commonly known as thermal throttling.

In Fig. 5 we observe how, by disabling the fan of the XU3, a large amount of variation gets introduced after some time running benchmarks. Contrary to the expected behavior of a progressive increase of execution time as temperature increases and processor frequency is restricted, we find that there are many reductions in execution time as well. By looking closely into this issue, we discovered that some cores were

[1] Some benchmarks contain a fixed number of repetitions, and others have been limited to a certain global elapsed time. This is specified in the label of the x-axis of each graph.

[2] Each value of the graph is the average of the previous 100 measurements, reducing noise for better readability and ease of comparison.

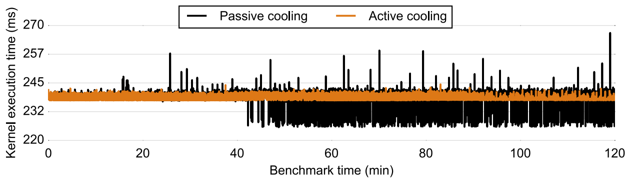


FIGURE 5. Performance variability due to cooling on XU3 (OpenCL).

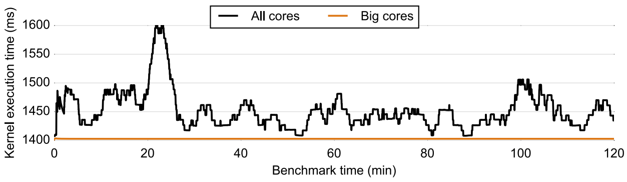


FIGURE 6. Processor affinity performance on XU3 (C/C++, window^2).

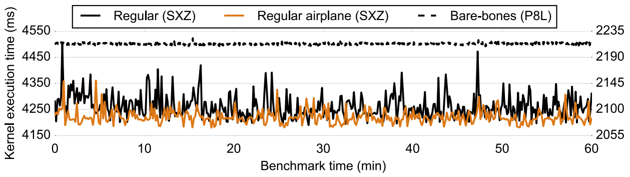


FIGURE 7. Performance variability between background process settings (C/C++). Y axes adjusted to show a proportional interval size across devices. Left Y axis labels correspond to SXZ.

being shut down intermittently. Our best guess is that this resulted in an increase of memory frequency or some other component beyond our ability to measure or control, due to an instant reduction of the power draw and consequent increase of headroom to the power cap. Our efforts trying to replicate this behavior by manually reducing processor frequency or switching cores off failed to show this behavior.

Again, different devices and benchmarks can make the effect of cooling vary greatly. It tends to take a large set of benchmarks to demonstrate this effect, so shorter runs are not as affected if devices can cool down between benchmarks. Automating this, however, provides the benefits we discuss in Sect. IV.

C. PROCESSOR AFFINITY

Many modern SoCs used as the computing platform for popular mobile devices are built around an Arm big.LITTLE multicore CPU. In its simplest form, which we find on the XU3 and P8L devices, the CPU is formed by two clusters of cores; one of them containing high-performance cores, and the other containing low-performance high-efficiency ones. By default, the OS handles the mapping of processes to CPU cores, and this is done by considering the computational load of these processes. A process that runs computationally intensive tasks will be moved to big cores, whereas others will run on little cores to improve battery life.

As this is done at runtime, processes are continuously migrated from one cluster to another as their behavior changes. When benchmarking, it is important to control the cluster where the execution takes place. Otherwise, results might vary depending on the heuristic decisions made by the OS about the benchmarking process. Periods of low computation, like those used to reduce temperature between repetitions or synchronization barriers, will make the OS migrate this process to little cores as well. As a result, when execution starts it may take an undetermined amount of time for the OS to migrate the process back to big cores and a context change will happen. Manually setting the benchmarking process affinity to the desired cluster gets rid of this variability, as it is shown in Fig. 6.

D. BACKGROUND PROCESSES

Even though Linux systems are inherently multitasking and there are always multiple independent processes running in the same machine simultaneously, it is frequently possible to ignore their impact on benchmark results as their influence is minimal and can be modeled as random noise. Android, on the other hand, adds a new layer of process scheduling with background processes and activities. They tend to add a greater overhead, via frequent network communications, and the amount of these will depend on the number of applications installed in the device. Fig. 7 shows that if a mobile device is only used for benchmarks, and all non critical applications and services are removed, execution time stability increases significantly. The *airplane* mode reduces the interference of network connectivity related background processes, but results are still not as reliable. Their estimated coefficients of variation [3] are, for the regular and airplane mode benchmarks, 1.06% and 0.54% respectively, whereas having close to no background processes running brings it down to 0.06%.

IV. TECHNIQUES TO REDUCE ERROR IMPACT

In Sect. III we presented a set of error sources that must be considered in order to obtain reliable results from benchmarking. We also determined features that make the impact of these error sources to be significantly reduced. However, some of these features cannot generally be considered available or easily achievable on any device, and indirect methods have to be used instead. Section IV-A presents indirect methods for dealing with the identified error sources during the benchmarking process.

Furthermore, even when all possible error sources are reduced through careful experiment design, random variations can introduce a different set of problems. If they are not properly managed, misleading conclusions might be drawn with not enough data to support them. In Sect. IV-B we show ways in which the reliability of results can be improved independently of how measurements are taken. Other considerations are discussed in Sect. IV-C.

[3] Measure of dispersion defined in terms of the sample standard deviation (s) and the sample mean ($\bar{x}$): $\widehat{c_v} = \frac{s}{\bar{x}}$

A. MITIGATING SYSTEMATIC ERROR DURING BENCHMARKING

1) PROCESSOR FREQUENCY

Switching frequency governors is only possible on rooted devices. This quite significantly limits the range of devices where it can be done, due to the reasons explained in Sect. II-A. Additionally, the *performance* governor is not always available, so other ways of fixing the frequency have to be found. Our XU3 and P8L devices did not have *performance* GPU governors so, in order to fix the GPU frequency to its maximum value, the minimum frequency parameter of their default governor was set to it. This has the limitation that it still requires root access and it must be set periodically because, when temperatures rise, it gets reset to default.

On devices with no root access, the only way to force an increase of processor frequency is to "trick" the governor into thinking that a compute-intensive task is running before starting the benchmark, and to make sure of keeping it that way along its whole duration. This strategy will only work if the default governor responds to high processor usage by increasing its frequency, unlike some power saving ones. Monitoring the relevant processor's frequency while some dummy workload is executed until maximum frequency is reached helps reduce the added overhead of this method. Fig. 8 and 9 show the performance difference between triggering a frequency increase before each run, if it is not already set at maximum frequency, and not doing so.

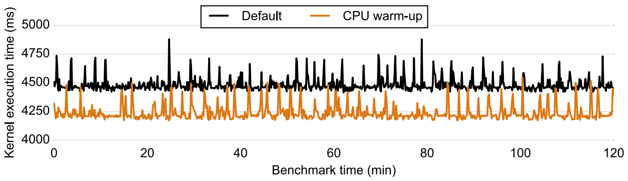


FIGURE 8. CPU frequency manipulation on SXZ (C/C++).

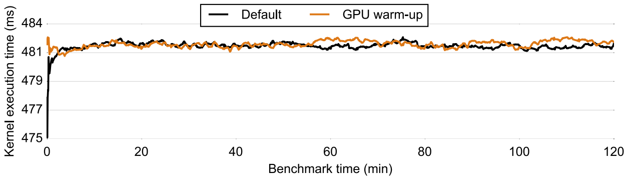


FIGURE 9. GPU frequency manipulation on SXZ (OpenCL, window^2).

In this case, we find that indirectly inducing an increase in CPU frequency results in an average relative runtime improvement of 5.39%. This tells us that, by default, the CPU is running this benchmark below its maximum capacity, resulting in a higher potential of result variability when changes in frequency happen organically. Our method still has limitations, because it cannot consistently keep frequency stable, but it makes big swings in performance less likely.

On the other hand, GPU results show a completely different picture. They indicate that GPU frequency stays mostly at the same level regardless of our tries to increase it before each run. Although there is not a significant global difference, we find the first few iterations tend to achieve a slightly better performance. These are dropped by warm-up code, so they are only shown on the baseline benchmark. While it tells us that there is actually potential for more performance, our results being skewed in both cases, there is no way to sustain that performance level for long enough to produce stable results without control over the GPU governor. We believe that, in this case, it is preferable to avoid the initial spike in performance to increase the precision of results.

2) TEMPERATURE

In terms of temperature, the vast majority of hand-held devices are only passively cooled. Extensive hardware modifications of the device would be needed in order to properly install active cooling elements onto one of these, which is extremely impractical and can result in permanent damage to the device. Furthermore, certain workloads may induce a larger power output than the thermal solution can effectively dissipate. We propose, instead of restricting benchmarking to development boards with active cooling, to introduce manual thermal management into the benchmarking framework regardless of the cooling solution.

By using internal temperature sensors, we can monitor temperature within the benchmarking framework avoiding making instrumented runs when temperature rises over a certain threshold. We effectively shift the responsibility of keeping temperatures at an acceptable level to our benchmarking system, instead of letting the OS deal with it by reducing performance during execution. This splits the benchmarking process into sections of intensive computation and sections of idle wait for the device to cool down.

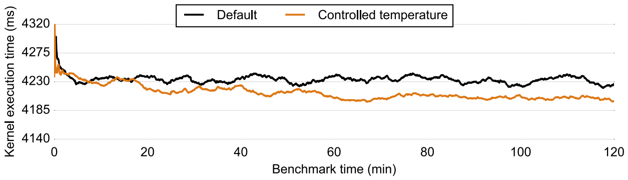


FIGURE 10. Temperature control on SXZ (C/C++, window^2).

Fig. 10 shows the effect of manual cooling as opposed to ignoring temperatures while benchmarking. In this case, we clearly see how performance diverges as more tests are carried out. Temperature-related performance behavior is greatly dependent on environmental conditions (i.e. ambient temperature), the hardware properties of the device in terms of processor power consumption and thermal dissipation, and the code that is benchmarked. All of these factors can make a significant difference in the effectiveness of this approach, but it is always an improvement over letting the OS change performance parameters during benchmarking.

3) PROCESSOR AFFINITY

Processor affinity of processes in big.LITTLE CPUs can be changed in Android through the `sched_setaffinity`

[20] Linux system call. If for some reason this call failed or was disallowed, the only possible way to achieve a similar behavior would be to add a compute-intensive warm-up execution right before running measured benchmarks. This would be done in an effort to indirectly let the OS know that this is a compute-intensive phase of the application, just as forcing an increase in processor frequency requires.

4) BACKGROUND PROCESSES

Even though, ideally, having dedicated devices for benchmarking stripped of any non-critical applications is the best way for reliably obtaining results, it is still important to find ways of achieving similar results using personal devices. This would allow any application developer to benefit from better insights, and significantly lower the preparation work that setting up a device for benchmarking takes.

The problem we want to solve is the overhead that Android background processes can unpredictably have over benchmarks. Given any hand-held device, we should devise a method to prevent any installed applications from interfering with the benchmark. Ideally, running the benchmarking underlying Linux process in real-time mode would result in it having maximum priority over the rest at the OS scheduler level. However, it is disallowed due to the implications in terms of "fairness" this would have on a mobile environment. Fortunately, Android provides a Java API [21] that can be used to kill other Android processes. By killing all non-critical processes before benchmarking we should be able to get a more stable environment to help with more reliable results. However, many of these processes will launch again shortly or sometime after being killed, adding a great deal of overhead at an unpredictable moment. Additionally, system signed applications which are non-critical cannot in many situations be killed by either the user or the benchmarking application via the aforementioned API. This is another limitation of this approach, for which the only solution is to obtain root access to the device.

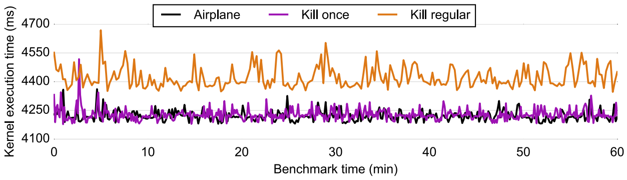


FIGURE 11. Background processes overhead control on SXZ (C/C++).

In Fig. 11 we see how stopping background processes once before starting the benchmark and stopping them in regular intervals compare to leaving them running. Since killing background processes results in a significant overhead when these are restarted, we find that this approach is, in this case, more than a 4.5% slower than the other two approaches. It also doubles their coefficient of variance, so it is clear that the benefits of forcing a lower amount of running background processes are well below the problems it introduces.

The comparison between killing background processes once before starting the benchmark and not doing so is much less clear. What we find is that the latter provides a slight advantage on precision (0.62% estimated coefficient of variation, versus 0.71%) and a minimal improvement on average runtime that is well within margin of error. On the other hand, by killing background processes we are able to execute 34.86% more benchmark runs during the same time interval.

This tells us that minimizing the amount of background processes running before starting the benchmark does at least help increase the performance of the whole system. This could be due to a reduction in global power consumption, which results in a lower thermal output and a less frequent need for device cool-down.

B. IMPROVING THE RELIABILITY OF RESULTS

After designing ways to increase the precision and remove the bias from benchmark measurements, there are still some questions that need to be answered about how these measurements should be processed and presented, enabling informed decisions and deeper insights. Additionally, we need to be able to determine how many repetitions of a benchmark are needed to reach the desired precision. This has been the focus of previous works on other platforms [22], but many of their recommendations apply in this context as well.

It is common practice to fix the number of repetitions of a benchmark in advance, but the required number of repetitions varies depending on the device, the algorithm and, in some cases, even the inputs. We propose data-dependent metrics, such as coefficient of variation and relative width of a confidence interval, as the stop condition. They allow testers to indicate the precision they require, and benchmarks should run only as many times as needed.

By following the procedures we presented in Sect. IV-A, we not only increase the probability of obtaining representative values, but also increase the stability of these measurements. This means that error tends to approach zero with more executions, as opposed to what would happen if factors such as temperature were not controlled. This property helps the mentioned stop conditions finish quicker. Of course, there is still a chance of never reaching the intended precision.

One way in which this can happen is by not being able to make measurements precisely enough. If the signal to noise ratio of the measurements taken is low enough, it may not be possible to ever reach the desired precision. For this reason, time limits to the benchmark should be added, so they are guaranteed to finish. In addition, outliers happening during the first repetitions due to JIT compilation or other issues, could heavily increase the required number of repetitions to achieve the precision goal. That case can be easily addressed by using a fixed amount of the latest measurements (sliding window), instead of the complete set, as input to the desired stop condition.

In the general case, however, outliers detected anywhere during a benchmark should be properly managed. As we know, each benchmark will run for a certain amount of repetitions without changes to either the environment, the measured code or the input parameters. Hence, outliers do not demonstrate a behavior worth analyzing, but only cases where outside factors have influenced the benchmark. Correctly detecting and removing these will result in an increase of precision, so we believe it is worth doing in this situation.

There is a large amount of ways to detect outliers on univariate data sets such as the ones produced by benchmarks [23]. Among them, some assume normality in the way values are distributed and others are robust even where that is not the case. By removing the main systematical error sources from benchmarks, we effectively increase the relative impact of smaller random errors, leading to normal distributions. Otherwise, we can find multi-modal distributions where each mode corresponds to a different performance level or processor frequency, and highly skewed distributions where the longer the benchmark runs the lower the performance is.

Taking into account these different cases, we believe that options like Tukey's method [24] should be preferred to others like Z-Score [25], due to their higher robustness against not normally distributed data and masking problems that happen when an outlier hides the presence of others. Histograms, too, are powerful tools to understand the results of a benchmark. We recommend using a sliding window method to adapt outlier detection to the behavior of the system and reducing computational overhead, as well.

In terms of reporting results, minimum and maximum values are not as robust as metrics such as the median or quartiles. Where the first are more sensitive to random noise and outliers, the former are much more stable metrics that should be preferred. Additionally, precision metrics like the standard deviation or confidence intervals must be reported and used for comparison purposes. Otherwise, wrong conclusions might be extracted where apparent differences in performance or energy are in fact owing to lack of precision. The histogram, too, is a powerful tool to understand the results of a benchmark.

C. ADDITIONAL CONSIDERATIONS

There are other factors that need to be taken into account in order to achieve more reliable benchmark results, which complement what has been presented in Sect. IV-A and IV-B:

- Running benchmarks shortly after booting up the OS could result in an irregular overhead over time, due to start-up processes running on the background. Waiting until the device stabilizes helps increasing benchmark precision.
- Battery powered hand-held devices should be connected to external power to avoid frequency governors from going into more power-saving states during benchmarking.
- When benchmarking performance or energy consumption of an Android application, it must be ensured that a *release* build with no debugging support is used. Additionally, tools provided by the Android Studio IDE like *Instant Run*, targeted at quickly replacing code in execution, must also be disabled if benchmarks are launched from this environment. These, if incorrectly managed, will place overhead on the benchmarked code that will significantly skew the results.
- The execution time of the code being measured must be large enough that the impact of the benchmarking code that is launching its execution, and the precision of the meters being used, are not significant. This can be achieved by running the code multiple times per measurement and dividing each measurement by that number of repetitions.
- Depending on the compilation method used for Java code on particular versions of the OS, sometimes it might be necessary to run multiple repetitions of a benchmark before making any measured runs. These warm-up iterations are commonly used to benchmark JIT-compiled languages. In our testing, we were not able to measure the impact of this factor, but we believe it is still worth noting, as we limited our evaluations to kernels of a small size that could have been JIT-compiled quickly without a noticeable overhead.
- Ambient temperature will have an effect on the performance behavior of benchmarks, even when manually ensuring the device temperature stays within a working range before making instrumented runs. Keeping ambient temperature constant and reporting its value can yield reproducibility improvements.
- Forcing the release of memory used by the benchmark after each run, as well as the Java garbage collector, helps create a more controlled testing environment. The probability of garbage collection passes during the execution of a benchmark is greatly reduced, and memory usage stays constant across iterations. Previous works [26] go even further, recommending re-randomizing the memory layout of code, stack and heap objects at runtime in order to ensure normally distributed results.
- If manually controlling temperature at the same time as warm-up runs are used to force processor frequency up, the interaction between both must be considered. Cool-down has to be done before the following warm-up, and the period of cool-down needs to bring the device to a lower temperature level than ordinary to account for a following warm-up process. The lower threshold of temperature should be tweaked such that there is enough headroom to benchmark several repetitions after warm-up without hitting the upper threshold again. Otherwise, most of the time will be spent managing temperature and frequency, and not running benchmarks.
- Warm-up runs intended to increase processor frequency must be bounded, so that they do not produce the opposite effect: Increasing temperature until thermal protection mechanisms are triggered.
- Device temperature, frequency governors and processor frequency data in Android is accessed through

device-dependent paths in the underlying Linux file system, so implementing many of the proposed methods require some amount of previous exploration of the device in which benchmarks will run.
- Sensor data is expressed in varying units among devices, and some sensors always return invalid values. Currently, it is important to check these features and make sure to use the right sensors on each device. A standardized cross-platform API to query these sensors would be of great utility in this context.
- Temperature and frequency-controlling processes add a considerable overhead to benchmarks in terms of total required time to run a certain number of tests, as we show in Sect. VI. In some situations, we may discover that one or more of these processes are not making a noticeable impact on reliability, and more repetitions could have a greater benefit than granular control over the environment where they run.
- Starting in Android 9.0, new application power management features allow users to restrict certain applications from running in the background. Application developers looking to execute a benchmark with no root access could benefit from restricting every other application in the system to reduce interference.

V. RANCID FRAMEWORK

A. RELIABLE BENCHMARKING FRAMEWORK DESCRIPTION

Implementing the large set of systematic error prevention strategies presented in Sect. III and IV in an ad-hoc way for each benchmark would be a largely impractical task. We believe these strategies should be benchmark-agnostic and independent from the code to evaluate. For that reason, we have designed and implemented a modular and extensible benchmarking framework that aims to provide a simple and powerful reliable benchmarking interface to application developers[4]. Its goals are to split application logic from the benchmarking methodology, to bring a simple and very flexible interface to developers for obtaining, processing and exporting the metrics relevant for their benchmarks, to automate and hide the complexity of making reliable measurements and designing computational experiments on mobile platforms, and to provide the infrastructure to test and compare novel benchmarking methodologies on any platform supported by Java.

The design of this framework is very flexible so that it can accommodate benchmarks with particular requirements of any kind, as well as extensible to allow users to build on top of it and easily get new features integrated. At the same time, it provides a rich set of default components, which can be built upon over time, allowing the creation of most benchmarks with very little effort. Flexibility is important because it is impossible to enumerate all the methodologies that could potentially be used to do the experimentation. Furthermore, we want to make other benchmarking methodologies possible to achieve within the same framework. This way, applications can swap between them seamlessly and direct comparisons between methodologies can be obtained. For example, applications that monitor system parameters and change their behavior depending on them could not be directly evaluated using our methodology if these parameters coincide. In these cases, conflicting aspects of the methodology could be omitted or, more importantly, the ability of setting system parameters could instead be used to evaluate the application in various specific circumstances.

Other frameworks with similar objectives are Google Caliper [13] and OpenJDK's Java Microbenchmarking Harness (JMH) [12]. These frameworks provide annotation-based APIs to developers, helping reduce the learning curve. *Rancid*, on the other hand, aims at a higher level of configurability without compromising on usability, so it exclusively uses an object-oriented interface. It provides dynamic control over the benchmarking process, through its ability to modify its runtime behavior and set up dynamically produced parameters. It can be used as a platform on top of which to build more feature-rich benchmarking tools, without losing the ability to be tailored to specific platform conditions in order to provide the most reliable measurements. Both of these alternatives have very rich backends that could implement some or all of the ideas in our methodology, but this is far from straightforward, since allowing that type of customization was not a main consideration in their design. Whereas Caliper and JMH focus on solving the issue of avoiding common microbenchmarking pitfalls, *Rancid* focuses on providing flexibility in terms of how these are executed and how the underlying system and hardware is manipulated in order to ensure a stable execution environment. Furthermore, *Rancid* has been designed with issues on mobile environments at the core, but the alternatives are focused on classic architectures and their integration in Android is not straightforward.

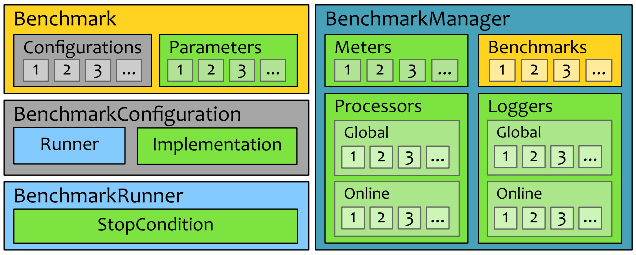


FIGURE 12. Main components of the *Rancid* framework.

Fig. 12 shows the basic structure of our *Rancid* framework. Its main functions are separating the way benchmarks are executed and evaluated from their implementation, and providing a platform on top of which benchmarking methodologies can be tested, evaluated and compared, and swapped easily depending on which the fittest is on every environment. Developers only need to provide implementations and parameters, as well as setting up how they would like the benchmark

[4] `https://github.com/HPC-ULL/Rancid`.

execution to be carried out, and our framework provides a large set of reusable and extensible tools that greatly reduce the work required to obtain and analyze reliable metrics from repeated executions of a piece of code. The main components of our framework are the following:

- **BenchmarkManager:** Manager class able to handle and run several benchmarks, and report aggregated *Results* back. It is composed of a set of *Benchmarks*, *Meters*, *ResultsProcessors* and *ResultsLoggers*. Online processors and loggers run after each combination of *BenchmarkConfiguration* and *Parameters*, whereas the global ones only run after each completed *Benchmark*.
- **Benchmark:** It contains a set of *BenchmarkConfigurations* and *Parameters*. Each combination of these is executed and processed by the online loggers and processors of the parent *BenchmarkManager*, where results are aggregated as well.
- **BenchmarkConfiguration:** A configuration is a pair of *BenchmarkRunner* and *BenchmarkImplementation*. It ties implementations of a benchmark with the way in which their execution should be handled.
- **BenchmarkRunner:** Runner class that is able of managing the execution of any single *BenchmarkImplementation* given a certain set of *Parameters*. Its function is to handle external factors that could impact results.
- **Meter:** Measurement class used in order to obtain metrics from running benchmarks. Users could create their own *Meters* to measure any relevant property of the benchmarked code.
- **ResultsProcessor:** Analysis interface for *Results*. It can be used to calculate averages, histograms or any other type of metric from data produced by *Meters* and previously run *ResultsProcessors*. It has the ability to read and modify *Results*, so apart from adding new nodes it can modify or delete them. This is useful for discarding outliers or warm-up runs. Each analyzer processes the complete set of *Results* specified, depending on if it is working online or globally, in the order they were added to the parent *BenchmarkManager*.
- **ResultsLogger:** The *Results* logging interface. Its purpose is to output benchmark results as they are produced, always right after analyzers have processed them. Their output can be directed to a file for further processing.
- **Parameters:** A set of parameters, intended to be used in a benchmark run. The user defines the names and values of each parameter in a parameter set, and the corresponding implementation should retrieve and use them as input. They must not be modified, so that further executions do not result on different outcomes or behavior.
- **BenchmarkImplementation:** The implementation of a benchmark, or the code that is to be benchmarked. Users define their implementation by indicating how to set it up from a set of parameters, how to finalize it and what the code to instrument is. This way, we avoid as much as possible measuring anything else than the developer wants to, and we allow using any regular code from the application without changes.
- **StopCondition:** Interface for classes representing stop conditions, or logic for deciding whether to keep running a certain benchmark or stop. As each combination of *BenchmarkConfiguration* and *Parameters* is executed repeatedly, we need to indicate how many times this is done. Stop conditions may be based on total runtime, number of repetitions, confidence interval width or any other property of the partial results.
- **Results:** Results for a benchmark are stored in a recursive hierarchical structure based on nodes which can be lists, key-value maps or single values. By default, the levels used are *MultiBenchmark*, *Benchmark*, *ParameterSet*, *Metric* or *Analysis*, and *Value*.

Users are expected to create their own *BenchmarkImplementation* subclasses, and often define their own *Meters*. Other main components like the *BenchmarkManager*, *Benchmark*, *BenchmarkConfiguration*, *Parameters* or *Results* should be usable without modification for any purpose. In certain cases, users may want to define their own *ResultsProcessors*, *StopConditions*, *BenchmarkRunners* or *ResultsLoggers*, when they need functions that are not already present in our framework. Some of these may include logging results in a certain format, calculating specific metrics from benchmark results or preventing errors by considering external factors during benchmarking.

However, our framework already provides a large number of predefined components that reduce the likelihood of developers to have to implement their own. They can select the components they need to design their benchmarks, easily replacing them as they discover new needs or limitations of their approach. Some of these components, which can be extended in the future, are:

- **BenchmarkRunners:** *ModularBenchmarkRunner*. This runner can be configured to execute any list of actions in any order specified, separated by actions to run before each benchmark, before each repetition, after each repetition and after each benchmark. Apart from the provided actions (*PrintAction*, *SleepAction*, *WarmUpAction*, *CoolDownAction*, *FileWriteAction*, *CompositeAction*, *ConditionalAction*), users can easily define their own.
- **Meters:** *ExecutionTimeMeter*, *SucessfulRunsMeter*, *FileContentsMeter*. These meters are provided to easily get started, but care should be taken in order to measure the right value. Execution time meters should be added last to the list of meters, so they do not measure the overhead added by other meters.
- **ResultsProcessors:** *ArithmeticAverageAnalyzer*, *MaxAnalyzer*, *MinAnalyzer*, *StdDeviationAnalyzer*, *HistogramAnalyzer*, *SumAnalyzer*, *WarmUpIterationsFilter*, *WindowIterationsFilter*, *InvalidRunsRemover*, *ResultsRemover*. Some processors are designed to read results and produce new nodes which are added to the global

Results object, whereas others carry out the task of replacing or removing nodes.

- **ResultsLoggers:** *HumanReadableResultsLogger*, *JsonResultsLogger*, *XmlResultsLogger*. The human-readable results logger is designed to be used as an online logger, showing results as they are obtained. The others can be used to generate data files to be further processed or plotted.
- **StopConditions:** *ErrorStopCondition*, *FixedIterationsStopCondition*, *ElapsedTimeStopCondition*, *{And, Or, Negate}StopCondition*. We provide common stop conditions and a set of logical operators to combine them and construct more complex expressions.

B. USAGE EXAMPLE

At a minimum, in order to use our *Rancid* framework, the user only needs to create the implementation of the code they want to benchmark and set up the desired components around a single *BenchmarkManager*. Fig. 13 illustrates how a user would implement a Gaussian Blur benchmark based on their own implementation of this algorithm.

```java
public class JavaBlurImplementation
    extends BenchmarkImplementation {
  private GaussianBlur alg;
  private int imgId;

  public JavaBlurImplementation () {
    super("Java Gaussian Blur");
  }
  // Called per benchmark
  public void setupBenchmark (Parameters p) {
    int radius = p.getParameter("Radius");
    this.imgId = p.getParameter("ImageID");
    this.alg = new GaussianBlur(radius);
    alg.buildKernel();
  }
  public void finalizeBenchmark () {
    alg.freeMemory();
    this.alg = null;
  }
  // Called each repetition
  public void initParameters () {
    alg.setInput(imgId);
    alg.decodeInputBitmap();
  }
  public void instrumentedRun () {
    alg.run();
  }
}
```

FIGURE 13. Example implementation of a Gaussian Blur benchmark.

After defining the Gaussian Blur benchmark implementation, a developer would need to define the whole execution properties of the benchmark using provided or custom components, as shown in Fig. 14. This is a very modular approach that, once set up, allows easy tweaking and reuse.

Lines 9-14 define a condition that will stop when another two stop conditions both meet. One of these specifies a fixed number of iterations, whereas the other depends on the measured execution times to reach a certain coefficient of variation within the last repetitions. Lines 16-22 create a benchmark runner that uses the previously defined stop condition, and that adds a one second pause preceded by a message before running each benchmark. In lines 25-31, we create a new benchmark configuration from the runner we created above, and an instance of the Gaussian Blur implementation defined in Fig. 13. This is the only required configuration that we would need to add for additional benchmarks. If there were multiple implementations of the same algorithm we wanted to compare, we would create a configuration for each of these and add them to the same *Benchmark*.

Lines 34-42 create two sets of parameters used in the implementation of the benchmark. To be noted is that the names used when creating parameter sets must match the ones used to retrieve their values in the implementation.

```java
 1 final int warmup = 10;
 2 final int window = 20;
 3 final int iterations = 20;
 4 final double covThreshold = 0.05;
 5 OutputPrinter printer =
 6   new PrintStreamOutputPrinter(System.out);
 7
 8 // Configure runners and their stop conditions
 9 AndStopCondition stopCondition =
10   new AndStopCondition(
11     new FixedIterationsStopCondition(
12       iterations + warmup),
13     new ErrorWindowStopCondition(
14       ExecutionTimeMeter.TITLE, window, covThreshold));
15
16 ModularBenchmarkRunner runner =
17   new ModularBenchmarkRunner(stopCondition);
18 runner.addAction(BenchmarkStage.PRE_BENCHMARK,
19   new PrintAction<Void>(printer,
20     "Sleeping before benchmark..."));
21 runner.addAction(BenchmarkStage.PRE_BENCHMARK,
22   new SleepAction(1000));
23
24 // Create benchmark configurations and benchmark
25 BenchmarkConfiguration javaConfig =
26   new BenchmarkConfiguration(runner,
27     new JavaBlurImplementation());
28
29 Benchmark blurBenchmark =
30   new Benchmark("Gaussian Blur Benchmark");
31 blurBenchmark.addConfiguration(javaConfig);
32
33 // Create parameter sets
34 Parameters params = new Parameters("FullHD");
35 params.addParameter("ImageID", R.drawable.fhd);
36 params.addParameter("Radius", 5);
37 blurBenchmark.addParameter(params);
38
39 params = new Parameters("4K");
40 params.addParameter("ImageID", R.drawable.uhd);
41 params.addParameter("Radius", 5);
42 blurBenchmark.addParameter(params);
43
44 // Create and configure benchmark manager
45 BenchmarkManager mgr = new BenchmarkManager();
46 mgr.addBenchmark(blurBenchmark);
47
48 mgr.addMeter(new SuccessfulRunsMeter());
49 mgr.addMeter(new ExecutionTimeMeter());
50 mgr.addRunProcessor(new InvalidRunsRemover());
51 mgr.addRunProcessor(
52   new ResultsRemover(ResultTypes.Metric,
53     SuccessfulRunsMeter.TITLE));
54 mgr.addRunProcessor(
55   new WarmUpIterationsFilter(warmup));
56 mgr.addRunProcessor(
57   new ArithmeticAverageAnalyzer(
58     ExecutionTimeMeter.TITLE));
59 mgr.addOnlineLogger(
60   new HumanReadableResultsLogger(printer));
61 mgr.addGlobalLogger(new JsonResultsLogger(printer));
62
63 // Run benchmarks
64 Results results = mgr.runBenchmarks();
```

FIGURE 14. Example benchmark configuration.

These parameter sets are then added to the benchmark. Lines 45-61 create the benchmark manager to which the previously constructed benchmark is added, along with an execution time meter and a set of analyzers and loggers. Benchmarks start running when line 64 is executed, which returns the *Results* object for additional processing if needed.

By creating a custom *BenchmarkRunner* or adding the right actions to a *ModularBenchmarkRunner*, we can implement all the processes explained in Sect. III and IV-A to create a much more reliable benchmarking procedure on Android devices. Furthermore, ideas mentioned in Sect. IV-B can be trivially implemented as stop conditions or processors.

Since our framework is not dependent on particular Android features, it can be used for benchmarking code in any other system, so custom runners for reliable benchmarking on these other systems can be implemented as well. Although Java code is the easiest to instrument using this framework, implementations can be created in any language that can interface with Java. C/C++ is one of these cases, with the Java Native Interface (JNI) being the standard way of communication. Using native code as a bridge, other programming models can be benchmarked as well. We evaluated Java, C/C++ and OpenCL implementations in order to demonstrate the flexibility of this system.

VI. FRAMEWORK EVALUATION

A. SINGLE BENCHMARK EVALUATION

We evaluate the effectiveness of the *Rancid* framework, presented in Sect. V, by comparing execution times on 2 hour benchmark executions of a Gaussian Blur kernel implemented in Java, C/C++ and OpenCL on each of the hardware platforms presented in Sect. II-B, using a single 3840 × 2160 (4K UHD) pixels input image. For each of these cases, we run a managed version that implements all applicable error countermeasures as described in Sect. III and IV, and a baseline, which represents a naive benchmark execution taking no specific measures against error.

In this section, we use the concepts "precision improvement" and "performance improvement" to compare the results obtained on baseline and managed benchmarks. Precision improvement refers to the achieved reduction in the sample standard deviation, whereas performance improvement measures the reduction in median execution time obtained in each benchmark. Significant variations in performance mean skewed results, whereas variations in precision point to differing levels of background random noise. Ideally, our managed benchmarks should be able to eliminate any skeweness and significantly reduce random noise in order to obtain accurate and reproducible results. However, these are independent features and improvements in one do not necessarily have to reflect in the other. Improvements in any of the two metrics are beneficial, even if the other remains unchanged. The baseline itself does not always suffer from significant random or systematic error sources. In these cases, improvements to the corresponding metric are not expected.

It is to be noted that the performance improvements we find in our testing are not related to possible speedups of the mobile application that is being benchmarked. These improvements correspond to performance headroom available in the system that, if not properly managed, could prevent achieving reliable measurements and comparisons during benchmarking. The application developer's task is to use these reliably obtained measurements to inform application optimization decisions that end users will take advantage of.

Our managed benchmarks use different countermeasures against errors due to processor frequency, temperature, processor affinity and background processes depending on the capabilities of each device. Warm-up iterations, in cases where the frequency governor could not be changed or forced JIT compilation was necessary, were executed for up to 10 seconds or until processor frequency was at its maximum value. Temperature thresholds over which manual cool-down was introduced were chosen depending on the optimal working temperature range for each device, avoiding thermal throttling without spending too much time cooling down the device. Background applications were killed before starting each benchmark, and the benchmarking thread was scheduled onto high-performance cores in big.LITTLE architectures. None of these methods are applied to the baseline. Table 2 shows a summary of the results we obtained and compares the measured precision and performance of the baseline and managed benchmarking methods implemented in the *Rancid* framework.

TABLE 2. Summarized improvements on 2 hour benchmarks.

Device	Implementation	Performance improvement	Precision improvement
XU3	Java	0.802%	**98.476%**
	C/C++	0.218%	**99.658%**
	OpenCL	**35.396%**	**93.868%**
P8L	Java	0.117%	0.889%
	C/C++	0.038%	**-17.817%**
	OpenCL	**20.720%**	**96.139%**
SXZ	Java	0.737%	**35.025%**
	C/C++	0.610%	**49.359%**
	OpenCL	-0.070%	**32.060%**

These metrics allow comparing the effectiveness of benchmarking methods in terms of random noise and systematic error reduction. Imprecision in the measurements can preclude from extracting useful data, whereas skewness could lead to wrong conclusions. Fig. 15 gives a graphical overview of these results for ease of comparison. There, we can intuitively see precision differences by comparing whisker lengths and box sizes, and performance differences by looking at the relative positions of the median markers.

Results show high increases in the precision of results in the vast majority of cases. The outlier we find on the P8L C/C++ benchmark is due to having obtained very precise results in the baseline ($\sigma = 2.35$ms), though the added noise on the managed benchmark is not very significant ($\Delta\sigma = 0.42$ms). The Java case in P8L does not see great precision improvements due to the same reason.

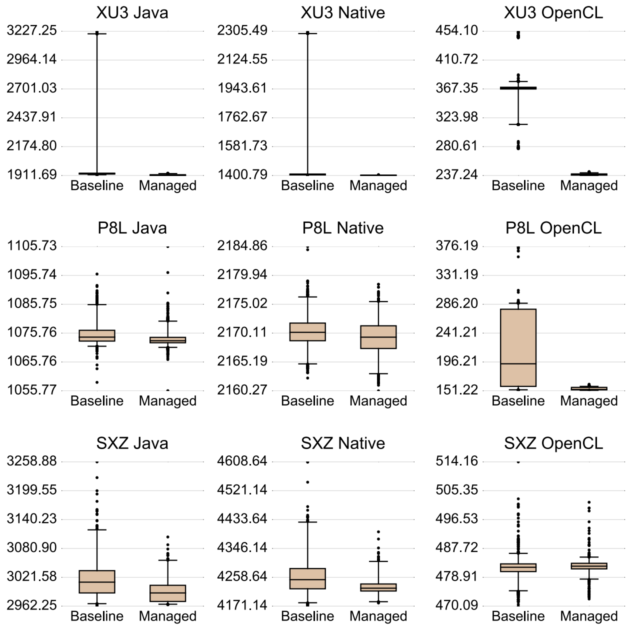


FIGURE 15. **2 hour benchmarks results. Each subplot is scaled to the minimum and maximum execution times of the baseline-managed pair. Boxes represent the first and third quartiles, along with the median, and whiskers extend to the 1st and 99th percentiles.**

The general increase of precision results in a much more robust foundation on top of which to achieve valuable insights, because a lower dispersion allows making finer grain comparisons between different results. With regards to skewness correction of measurements, results show much less drastic improvements.

This is to be expected, since major systematic error sources tend to be less common. The main reasons for treating them are avoiding edge cases and compounding of errors. OpenCL execution on the XU3 and P8L platforms saw major improvements in performance and precision mainly due to the behavior of their default GPU frequency governor, which tends to change frequency levels often and spend significant portions of the time on the lower levels.

Tests carried out in SXZ were impacted by temperature, as seen in Fig. 10, and warm-up iterations helped maintain slightly higher processor frequencies on average. On this device only countermeasures that do not require root access were used, showcasing again the effectiveness of the methods described in Sect. IV.

B. ERROR EVOLUTION EVALUATION

In practice, benchmarks of small kernels are rarely executed for very long periods of time, because it would be a waste of resources. Measurements and error of well-behaved benchmarks stabilize and, when they do, further measurements will not vary results. Previous results in this paper have focused on long executions that represent cases where many kernels, implementations or parameters are evaluated in sequence on the same system, so factors that increase their importance over time, like temperature, can be demonstrated and evaluated. Benchmarks, however, should only run for as long as necessary until the actual measurements have provided a certain confidence that they are representative.

We evaluated the evolution of benchmark error over time by comparing the running median execution time, obtained for an amount of repetitions ranging from 1 to 1000, to a *reference* representative value obtained as the median of 2 hours of benchmarking using our reliable methodology.

In addition, the inter-quartile range (IQR) evolution was analyzed in order to assess the precision of results as more repetitions are finished. Fig. 16 demonstrates the most representative behaviors we observed.

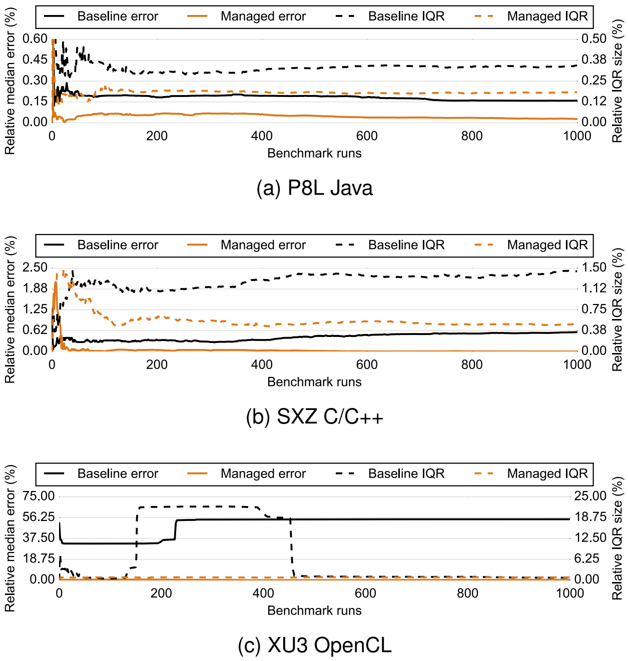


FIGURE 16. **Error and precision evolution graphs.**

In these plots we represent the relative error between the median execution time and the *reference*, and the IQR size relative to the magnitude of the measurement. These values are plotted for the baseline benchmark execution, as well as the managed benchmark execution.

A well-behaved benchmark should show, in these graphs, a progressive reduction of error until reaching zero and stabilizing when enough experimental measurements are made. It should also show IQR to decrease as more experiments are done, until it reaches a stable level that represents the random noise floor for that experiment.

The three basic behaviors we observed are represented by each of the sub-figures:

- Steady reduction of error over time that converges slightly away from the *reference* value. Fig. 16a is one instance of this very common behavior. Even though the baseline benchmark shows the type of behavior that was expected, if we compare it to the managed benchmark, we see that, after convergence, it suffers from a slight deviation from the *reference* value. It shows that some

TABLE 3. Reproducibility evaluation on 30 runs of 50 iterations benchmarks. Absolute and relative improvements over baseline are displayed.

		Performance improvement				Precision improvement			
Device	Implementation	Gaussian Blur		Pattern Thinning		Gaussian Blur		Pattern Thinning	
XU3	Java	9.384 ms	0.487%	-34.157 ms	-0.485%	**2.070 ms**	**43.669%**	**-4.752 ms**	**-8.582%**
	C/C++	1.813 ms	0.129%	29.565 ms	0.666%	-0.148 ms	-6.290%	**9.234 ms**	**32.025%**
	OpenCL	**76.807 ms**	**24.275%**	**160.809 ms**	**14.464%**	**18.092 ms**	**98.410%**	0.621 ms	56.114%
P8L	Java	6.246 ms	0.579%	**827.695 ms**	**17.666%**	0.408 ms	54.800%	**3.926 ms**	**75.413%**
	C/C++	**23.243 ms**	**1.060%**	**905.681 ms**	**11.933%**	**4.922 ms**	**87.498%**	**1.997 ms**	**38.692%**
	OpenCL	**122.550 ms**	**44.513%**	**129.033 ms**	**11.782%**	**25.322 ms**	**99.589%**	**1.346 ms**	**57.487%**
SXZ	Java	13.145 ms	0.437%	**79.849 ms**	**1.126%**	**3.955 ms**	**24.208%**	**12.432 ms**	**23.768%**
	C/C++	6.065 ms	0.133%	**252.235 ms**	**1.758%**	**15.482 ms**	**16.718%**	**119.684 ms**	**33.522%**
	OpenCL	0.099 ms	0.021%	0.565 ms	0.044%	0.286 ms	54.834%	**3.305 ms**	**81.084%**

systematic error source is present in the baseline. The precision of the managed version is also higher than that of the baseline.

- Error increase over time instead of decreasing, leading to divergence and impossibility to obtain reliable results. Fig. 16b shows how the baseline benchmark error and dispersion increase over time instead of going down and stabilizing. This is the type of effect that we expect temperature to produce on these platforms. Ignoring the impact of this problem makes it so that no amount of executions will yield reproducible results. On the other hand, the countermeasures we designed solve this problem in the cases we have studied.
- Quick convergence followed by a reduction in precision and performance, to finally converge further away from the *reference* value than initially. The behavior shown in Fig. 16c is a worst-case scenario of how some systematic error sources such as frequency governors can significantly skew results in a way that is not easily detectable. The baseline benchmark very quickly converges to a 35% error level over the *reference*, while maintaining a very low IQR, to then see a dramatic increase in IQR followed by an increase in execution time. It finally provides misleading results with a high level of confidence, after a final reduction in IQR, in contrast to our managed benchmark that quickly converges to a 0% level of error in this case.

Generally, we find that the managed benchmarking method is more reliable at reaching the *reference* value, while achieving a higher precision. It also consistently displays the expected behavior of a progressive reduction of error and IQR as more repetitions are executed, without significant increases after the first few repetitions. This shows that it is not enough to run tests more times in order to obtain more reliable and precise results, but a correct management of the intrinsic error sources of the system is required as well.

In terms of deciding what amount of repetitions is sufficient, what Fig. 16 shows is that a fixed amount or a target precision value are not enough. It appears to be preferable to run until both median execution times and IQR stabilize.

C. REPRODUCIBILITY AND RELIABILITY EVALUATION

The ultimate goal of our methodology is to improve the reproducibility of benchmark measurements on mobile platforms, helping add scientific rigorousness to these evaluations. Our results have shown the gains in precision and accuracy our methodology is able to achieve on individual benchmarks, but we need to assess its effectiveness in terms of producing the same results on separate independent repetitions of a benchmarking experiment as well.

To that end, we reproduce on each of our testing platforms a set of 30 independent benchmark executions, each corresponding to individual application launches, for our Java, C/C++ and OpenCL implementations of Gaussian Blur and Pattern Thinning [27] kernels, which run for 50 iterations each. The amount of iterations of each benchmark has been chosen as a more realistic limit that should be able to provide statistically significant results. We demonstrate that our methodology and framework help obtain more reliable results independently of the kernel being evaluated.

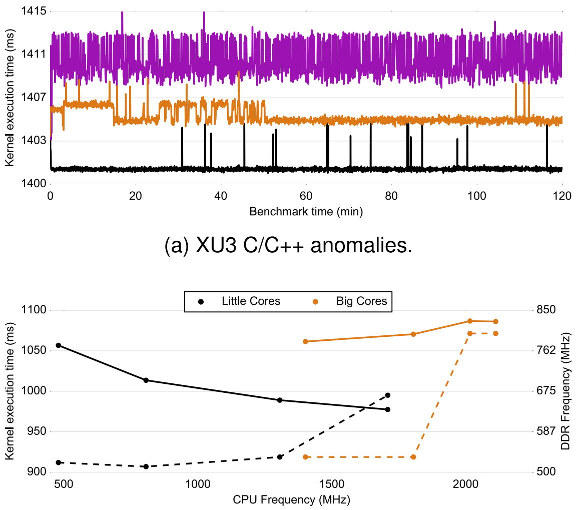
Table 3 summarizes the performance and precision improvements we obtained across benchmark executions. Values are aggregated from a single benchmark execution of 50 iterations, and those are further summarized across all 30 benchmark repetitions before being compared. The median and standard deviation of the median execution times on each managed benchmark execution are compared to the corresponding ones obtained in the baseline execution. In that table, we present relative and absolute improvements in performance and precision as reproducibility improvement indicators for Gaussian Blur and Pattern Thinning kernels, highlighting cases with over both *1 ms* and *1%* difference in performance or precision.

These reproducibility results show that our methodology improves the precision across most of the execution times measured during our benchmarking. This means that independent repetitions of a benchmark, given the same initial state, will report significantly more consistent values. We believe that, because these improvements are a consequence of a more stable system, most other metrics that can be extracted from these executions will be consistent as a result. At the same time, our methodology makes some significant improvements to performance, due to a reduction of the weight of the systematic error sources present in the system. We observe this improvement especially on the Pattern Thinning kernel, which is more compute intensive. This improvement comes mainly from the ability of fixing processor frequency and using performance cores on big.LITTLE architectures. However, in the SXZ case, where none of these options are available, we manage to get marginal improvements that increase with larger kernels. Precision improvements, however, are still very significant for most of our benchmarks.

Benchmarks running on the XU3's CPU, however, show that our methodology can, in some cases, introduce new sources of error that revert any improvements we were able to get. Looking at the results of each benchmark execution, we found that our methodology significantly increases the precision within each execution, but the aggregated results have a larger variation across independent executions, reducing reproducibility. This is one of the anomalies we study in Sect. VI-D. There is still room for improvement with regards to these types of anomalies, but we believe the advances we have achieved are an important first step towards improving the current state of benchmarking on mobile architectures.

D. ANOMALIES

Despite our best efforts to control every system parameter with a significant impact on benchmark results within our methodology, we have found evidence of other unidentified error sources, as well as counterintuitive behaviors we have not been able to explain. We illustrate these in Fig. 17.



(a) XU3 C/C++ anomalies.

(b) P8L OpenCL anomalies. DDR frequencies on dotted lines.

FIGURE 17. **Main anomalies found within our experimentation.**

Although our methodology significantly increases the precision of a single C/C++ Pattern Thinning benchmark on the XU3, we found that it also reduces the reproducibility of the obtained results when the benchmark is executed several times. After running several of these managed two-hour benchmarks we observed a few different behaviors that we summarize through three results in Fig. 17a. Many of the benchmarks showed the behavior of the black bar, in which there appears to be a baseline execution time and some outlier values that go over. However, the baseline was slightly different on each execution, taking values within a 15 ms range. In other occasions, the execution time would frequently jump between two different performance levels, which in very few cases would stabilize during the execution of the benchmark. Our observation is that execution times seem to fall within a few well-defined steps, like these that result from changing processor frequency. Given that processor frequency was fixed, we believe that memory frequency could be the culprit. As we have not found a way to measure or fix memory frequency in that device, we are not able to give a definitive answer to this anomaly.

On the other hand, evaluating the effects of manually setting the processor affinity while executing GPU codes on the P8L, we came across another unexpected behavior. In Fig. 17b, the average execution times for the Pattern Thinning kernel on GPU using OpenCL are shown for each available processor frequency together with the average memory frequency they used during each of these tests. What we find is that little cores seem to produce better results than big cores independently of the frequency they run at, but, at the same time, the higher the frequency they run at, the lower the execution time is. On the other hand, increasing clock speeds to big cores only results in a performance reduction, even if memory frequency gets significantly increased as a result. Furthermore, big and little cores on this device are the same model but behave differently at similar frequencies, which we can see at around the 1400 MHz mark. Given that, in the Pattern Thinning kernel, most of the CPU time is spent waiting for the GPU to finish executing a simple kernel and quickly checking a flag to decide if it should be executed again, the variation we observe among CPU configurations seems exceedingly large. Our guess as to why this anomaly exists is that there must be a driver or hardware-level feature that allows a lower latency of communication between GPU and little cores. By testing several other kernels, it would be possible to confirm if this behavior is application dependent.

VII. CONCLUSION

Benchmarking is a tool of great importance on the optimization of compute-intensive codes. Classical desktop and server platforms, coupled with relatively low overhead OS, tend to maintain a stable behavior during these types of evaluations. Mobile architectures, on the other hand, given their thermal and power constraints, and due to the differing runtime execution modes of the applications written for them and the design goals of their OS, show much more varying levels of performance. This increased noise level makes the precision of measurements significantly harder to improve to a level where small variations on benchmarked code can be detected.

Recent advances on SoC technologies have made their computational capabilities much greater, widening the types of applications they are able to interactively execute. Some of these heavier applications include image processing, computer vision and artificial intelligence kernels. Reliably measuring the performance of such kernels through benchmarking is increasingly important, but the current state of the art oversimplifies the required benchmarking procedure by implementing straightforward techniques that have previously proved sufficient on classical systems.

In this work, we have identified the main systematic sources of error on mobile platforms that must be properly treated in order to avoid misleading benchmark measurements. We have provided with direct and indirect methods of reducing the impact of these error sources, as well as other random errors, increasing the precision and reproducibility of benchmarks for any application on these platforms. Although there are still other errors that need to be investigated, our experimentation shows that our methodology greatly improves the precision and reliability of benchmarks, on long and short runs equally. We recommend running multiple independent benchmark executions, each of them until the metric to measure and precision converge. This way, anomalies such as the ones we found can be noticed, and their impact on reproducibility reduced.

We have implemented our methodology in a modular and extensible framework to make it very simple for application developers to evaluate kernels of their applications in a reproducible way. Our *Rancid* framework can be the foundation for achieving reliable results on any platform, by adding the particular error countermeasures associated to each of these platforms. Advanced statistical analysis methods can be included to easily extract relevant information from any desired metric gathered through benchmarking.

REFERENCES

[1] J. Vitek and T. Kalibera, "R3: Repeatability, reproducibility and rigor," *ACM SIGPLAN Notices*, vol. 47, no. 4a, pp. 30–36, Jun. 2012, doi: 10.1145/2442776.2442781.

[2] S. Hunold and J. L. Träff, "On the state and importance of reproducible experimental research in parallel computing," *CoRR*, vol. abs/1308.3648, pp. 1–15, Aug. 2013. [Online]. Available: http://arxiv.org/abs/1308.3648

[3] Android Open Source Project. (2019). *Identify CPU Hot Spots*. [Online]. Available: https://developer.android.com/games/optimize/cpu-profiler

[4] Microsoft. (2019). *Measure App Performance in Visual Studio*. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/profiling/

[5] JetBrains. (2019). *Profiling Tools*. [Online]. Available: https://www.jetbrains.com/help/idea/cpu-profiler.html

[6] C. January, J. Byrd, X. Oró, and M. O'Connor, "Allinea MAP: Adding energy and OpenMP profiling without increasing overhead," in *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Cham, Switzerland: Springer, 2015, pp. 25–35.

[7] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 63–74, Dec. 2013.

[8] V. Horký, P. Libič, A. Steinhauser, and P. Tuma, "DOs and DON'Ts of conducting performance measurements in java," in *Proc. 6th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2015, pp. 337–340, doi: 10.1145/2668930.2688820.

[9] J. M. Kim, Y. G. Kim, and S. W. Chung, "Stabilizing CPU frequency and voltage for temperature-aware DVFS in mobile devices," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 286–292, Jan. 2015.

[10] R. Aviv and G. Wang, "OpenCL-based mobile GPGPU benchmarking: Methods and challenges," in *Proc. 4th Int. Workshop OpenCL (IWOCL)*, New York, NY, USA: ACM, 2016, pp. 3:1–3:4, doi: 10.1145/2909437.2909441.

[11] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, Oct. 2007.

[12] OpenJDK. (2019). *Java Microbenchmarking Harness (JMH)*. [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/

[13] Google. (2019). *Caliper*. [Online]. Available: https://github.com/google/caliper

[14] P. Mochel, "The sysfs filesystem," in *Proc. Linux Symp.*, 2005, p. 313.

[15] The Linux Kernel Development Community. (2017) *CPU Performance Scaling*. [Online]. Available: https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpufreq.html

[16] Android Open Source Project. (2019). *ART and Dalvik*. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[17] A. Acosta and F. Almeida, "Towards a unified heterogeneous development model in Android," in *Proc. 11th Int. Workshop HeteroPar, Algorithms, Models Tools Parallel Comput. Heterogeneous Platforms*, 2013, pp. 238–248.

[18] A. Acosta, S. Afonso, and F. Almeida, "Extending paralldroid with object oriented annotations," *Parallel Comput.*, vol. 57, pp. 25–36, Sep. 2016.

[19] S. Afonso, A. Acosta, and F. Almeida, "High-performance code optimizations for mobile devices," *J. Supercomput.*, vol. 75, no. 3, pp. 1382–1395, Mar. 2019, doi: 10.1007/s11227-018-2638-5.

[20] *Sched_Setaffinity(2) Linux User's Manual*, Linux Man-Pages project, 5th ed., Oct. 2019.

[21] Android Developers. (2020). *ActivityManager API Reference*. [Online]. Available: https://developer.android.com/reference/android/app/ActivityManager#killBackgroundProcesses(java.lang.String)

[22] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal. (SC)*, New York, NY, USA: ACM, 2015, pp. 73:1–73:12, doi: 10.1145/2807591.2807644.

[23] S. Seo, "A review and comparison of methods for detecting outliers in univariate data sets," M.S. thesis, Graduate School Public Health, Univ. Pittsburgh, Pittsburgh, PA, USA, 2006.

[24] J. Tukey, *Exploratory Data Analysis* (Addison-Wesley Series in Behavioral Science). Reading, MA, USA: Addison-Wesley, 1977.

[25] B. Iglewicz and D. C. Hoaglin, *How to Detect Handle Outliers*, vol. 16. Milwaukee, WI, USA: ASQ Press, 1993.

[26] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically sound performance evaluation," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA: ACM, 2013, pp. 219–228, doi: 10.1145/2451116.2451141.

[27] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Commun. ACM*, vol. 27, no. 3, pp. 236–239, Mar. 1984.

SERGIO AFONSO received the B.Sc. and M.Sc. degrees in computer science from the University of La Laguna, San Cristóbal de La Laguna, Spain, in 2015 and 2017, respectively. He is currently a Research Assistant with the Department of Computer Engineering and Systems, University of La Laguna. His research interests include programming models for parallel computing, performance optimization and portability, and code acceleration on mobile architectures and high performance computing systems.

FRANCISCO ALMEIDA received the degree and M.Sc. degree in mathematics and the Ph.D. degree in computer science from the University of La Laguna, San Cristóbal de La Laguna, Spain, in 1989, 1992, and 1996, respectively. He is currently a Professor with the Department of Computer Engineering and Systems, University of La Laguna. His research interests include parallel computing, parallel algorithms for optimization problems, parallel system performance analysis and prediction, skeleton tools for parallel programming, and Web services for high performance computing and Grid technology.