UPCYCLING ANDROID PHONES INTO EMBEDDED AUDIO PLATFORMS

Victor Zappi

Northeastern University
Boston, MA, US
v.zappi@northeastern.edu

Carla Sophie Tapparo

Independent Researcher
Buenos Aires, Argentina
sophiecarlatapparo@gmail.com

ABSTRACT

There are millions of sophisticated Android phones in the world that get disposed of at a very high rate due to consumerism. Their computational power and built-in features, instead of being wasted when discarded, could be repurposed for creative applications such as musical instruments and interactive audio installations. However, audio programming on Android is complicated and comes with restrictions that heavily impact performance. To address this issue, we present LDSP, an open-source environment that can be used to easily upcycle Android phones into embedded platforms optimized for audio synthesis and processing. We conducted a benchmark study to compare the number of oscillators that can be run in parallel on LDSP with an equivalent audio app designed according to modern Android standards. Our study tested six phones ranging from 2014 to 2018 and running different Android versions. The results consistently demonstrate that LDSP provides a significant boost in performance, with some cases showing an increase of more than double, making even very old phones suitable for fairly advanced audio applications.

1. INTRODUCTION

With its billions of users, Android is one of the most widely adopted technologies existing today [1, 2]. Even the more affordable Android phones have CPU and memory specifications that compare with or even top those of many platforms commonly used by academics, researchers and creatives to design audio applications, including the Raspberry Pi[1], Bela [3] and the Daisy Seed board[2]. Yet, the mobile phone market is characterized by a constant evolution of both software and hardware, with new updates and models released frequently. Although current Android phones boast impressive technical specifications, they are often abandoned by users due to software incompatibility or the desire to own a newer, more advanced device. This consumeristic approach to technology creates significant environmental and ethical issues.

Firstly, the regular replacement of mobile phones contributes to a throwaway culture that values disposability over sustainability [4]. Electronic waste (e-waste) generated by discarded phones is a growing concern as it contains hazardous substances such as lead, mercury and cadmium, which can pollute the environment and harm human health. Moreover, this consumeristic approach to technology—that spreads way beyond mobile phones—also perpetuates a cycle of social and economic inequality [5]. Not everyone can afford to upgrade their technologies regularly, and the constant release of new models creates an unnecessary pressure to keep up with the latest trends, contributing to a sense of inadequacy and exclusion among those who cannot afford to do so.

In contrast to these unsustainable trends, this paper describes LDSP, a technology that enables extending the lifespan of older Android phones by repurposing them as embedded platforms for audio application development. LDSP provides an environment that allows developers to leverage the full potential of the phones' hardware and avoid the limitations imposed by Android's runtime environment. With LDSP, phones that would have otherwise become obsolete can be given new life, decreasing the need for the purchase of new programmable audio technology and reducing e-waste. We discuss the implementation of LDSP, its capabilities and how it can provide a significant boost in performance for audio applications. Additionally, we present the results of our experiment using an oscillator bank to compare the performance of LDSP with that of a typical Android audio app, addressing the open problem of effectively employing Android phones for audio synthesis and processing.

2. BACKGROUND

2.1. Technology, E-waste and Upcycling

In the field of audio and music hardware design, there is a larger issue at play regarding the obsolescence and progress mindset surrounding technological products. This is in part enabled by manufacturers seeking to drive trends and increase consumption [6] and strongly resonates with humans' innate curiosity and will to experiment. This mindset is economically and environmentally harmful, given the decreased lifespan of equipment and the increased production of e-waste [7]. This issue has underlying epistemological roots, where technology-based or electronic arts are tied to the notion of progress, 'new is better', and consumerism, which is unsustainable [4]. The process of creating new technologies and their discard negatively affects the land, water, air, and all living beings.

Among consumer technologies, mobile phones are notorious for their short product cycles, with an average use time of around two years [8]. In many cases, phones are replaced with newer models even if they are still functioning, as a result of the expiration of support for essential apps or the operating system itself. In the case of Android phones, open software can be leveraged to extend the life span of the device beyond the end of support and continue its intended. Examples includes: Lineage OS [3], a mostly open-source operative system based on Android and maintained by a large community of developers; /e/ OS [4] and iodé OS

[1] https://www.raspberrypi.com/

[2] https://www.electro-smith.com/daisy

[3] https://lineageos.org/

[4] https://e.foundation/

[5], both open-source mobile operative systems forked from or powered by Lineage OS that focus on privacy; or the Free Software Foundation Europe's initiative Upcycling Android [6] that intends on teaching the possibilities given by free and open software to improve user agency against technological obsolescence. However, the applicability of these solutions tends to depend on the specific model of the phone and support for functionalities can only be extended for a finite period of time. For example, one of the two LG G2 Mini phones used in this work to test LDSP (see Section 4) is running Lineage OS 14.1 that is equivalent to Android 7.0, while the last official Android version that specific model can run is 5.0.2. This extended the overall life span of the phone of two years, from 2016 to 2018—i.e., when support for Android 5.0.2 and 7.0 officially ended, respectively. Furthermore, phones frequently encounter significant damage to their screens, batteries, or other crucial components that render them unusable for daily use, irrespective of software support. Repairing such devices is increasingly expensive and intricate due to the miniaturized design of the products, the presence of glued-in parts, and the overall concept of *economical obsolescence*, where the cost of repairs often outweighs the value of the device itself [8]. Nonetheless, in many cases damaged/broken phones can still be turned into different but functional devices.

There is a tradition of technological disobedience found worldwide through different approaches that involve recycling and reusing materials, from *hackerspaces* [9] to Gambiarra [10] and others [11, 12]. These practices recover components and devices that would otherwise be discarded, finding novel uses for them by surpassing limitations in innovative ways, which resist consumerism and planned obsolescence practically. Such practices shift our understanding of when an object becomes useless or expendable if it is not related to its functionality or lack thereof, forcing us to think critically about how we use, buy, and discard technology.

Various art-related projects and instruments are specifically aligned with the upcycling of materials and the political implications of such endeavors. For instance, the Echo project [13] explores creative and alternative uses of outdated and damaged technologies, fostering an atmosphere where audiences can engage in critical and aesthetic debates surrounding the possibilities of these technologies, away from their intended purposes. Similarly, the Gatorra instrument [14] was created through a hobbyist approach to circuitry, repurposing electronic and non-electronic components to create a unique final product, emphasizing the autonomy of the creator and promoting innovative ways of engaging with hardware.

Certain composers and musicians, including Yasunao Tone, Nicolas Collins, and the group Oval [15], use the glitching and skipping of compact disks to generate new sounds, chance-based compositions, and indeterminate performances. Though their approaches to technology may differ, they share an interest in using seemingly broken technology to encourage novel sounds.

Other instruments, such as the Concentric Sampler [16], repurpose outdated technology, like floppy disks and floppy disk drivers, with additional circuitry that loops and uses time-based granular synthesis for live performances of lo-fi noise. The author of this project discusses their motivation for fostering creativity through physical limitations and misuse of audio technologies. Similarly, Disky [17] is a D.I.Y. USB turntable that utilizes the mechanical parts found in obsolete hard disk drives, providing an accessible, reliable, and low-cost project for audio control. In both cases, the authors emphasize their motivations driven against the novelty-driven discard of technology due to its current way of production encouraged by consumerism. They see their upcycling methodology as a creative way of dealing with technology that is considered obsolete while fostering creativity and community.

[5] `https://iode.tech/`

[6] `https://fsfe.org/activities/upcyclingandroid/` [Accessed on 2023/05/26]

2.2. Android Audio Programming

There have been various efforts to turn Android phones into platforms for audio processing and synthesis, with applications like Nexus [18] and MoMubPlat [19] using web technologies and Pure Data/libpd, respectively. However, *faust2api* appears to be the most comprehensive project to date, offering optimized Faust audio/sensors processing code and graphical user interfaces designed to explore the acoustic features of handheld devices [20].

Despite their capabilities, all these platforms and environments are limited by the Android audio stack, which consists of several layers and buffers that can introduce significant computational overhead and latency in audio processing and synthesis applications. As a result, it is difficult to achieve the satisfactory audio performance, especially on outdated phones. This problem has been known to audio processing and synthesis communities for some time, as discussed in research such as [21].

These issues arise from the structure of the Android application framework that allows for hardware-agnostic development, even for code written in native languages like Faust or C++ using the Android NDK. Such code has to pass through multiple layers of the audio stack before it can exchange samples with the audio driver in kernel space. These layers include the application layer, Android's mixer and the audio HAL, each of which introduces some level of buffering and scheduling that can increase CPU workload and cause inconsistent latency. Figure 1 represents the typical audio stack in modern Android app architecture. Another crucial detail emphasized in this figure is that developing a high-performance Android app often requires combining at least two programming languages: Java for the overall app structure and C++ for the performance-critical components. Additionally, in most cases, a graphical user interface (UI) is necessary, which, in modern Android development, is typically implemented using yet another language, Kotlin.

Researchers have proposed various solutions to address this problem, including the technique described in [1], which leverages the Exclusive Mode of the AAudio API introduced by Google in 2017 to bypass many layers of the audio stack. However, this solution is available only on relatively modern devices (running Android 8 or above). And, in general, more work is needed to fully address these issues in Android audio processing and synthesis, particularly for interactive applications.

3. LDSP

LDSP is an open-source cross-platform environment designed to enable developers to create native C++ audio applications for Android devices. Unlike traditional Android apps that run within the Android Run-Time Environment/Dalvik Virtual Machine, LDSP generates executables that are dealt with directly by the kernel and can directly access memory and hardware resources. Essentially, LDSP turns Android phones into generic Linux embedded boards, with the only requirement being that the phone is *rooted*. This

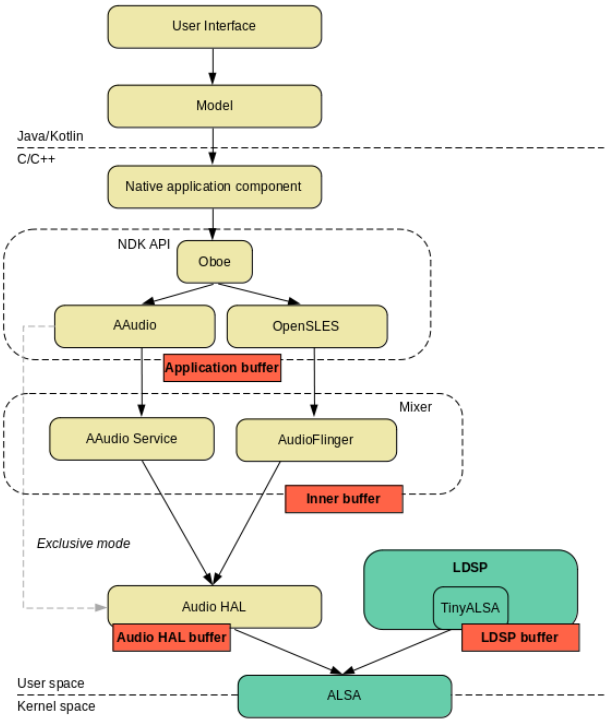

Figure 1: Audio stack comparison: LDSP stack (on the right) and modern Android audio app stack (on the left).

approach has a significant impact on code performance, as demonstrated later in this work. Currently, LDSP can be downloaded to a host computer, and developers can use it to cross-compile their native application and deploy it to their Android phone. The framework includes a C++ API, libraries and examples tailored to mobile audio development, with also direct access to the phone's sensors, buttons, touchscreen, LED lights and vibration motor. More details about the compilation process, libraries, features and examples can be found here [22], while the full source code is available at LDSP official GitHub repository[7].

LDSP is designed with portability in mind, and the low-level development and deployment workflow allow LDSP applications to bypass any resource allocation restrictions that standard apps may encounter. Moreover, LDSP is widely compatible across phones. Retro-compatibility with older Android versions is one of the most challenging aspects of Android development. The Android development framework is continually advancing, to support upcoming devices, better streamline general purpose app design and comply with security and privacy regulations. These are important issues, but contribute to the quick obsolescence of phones that are otherwise still functional. LDSP offers a solution to give new life to older devices regardless of the Android version they run or the hardware features they have. Thanks to the pure C++ implementation of LDSP, the same code can be built and run on most official Android versions, including phones with installed custom ROMs[8] based on Android.

At the core of the LDSP C++ framework lies a custom audio engine that is built around the TinyALSA library[9] and designed to directly control the Advanced Linux Audio Architecture (ALSA) kernel drivers. The audio engine provides an API to open any of the phone's capture and playback devices, synchronize them and set up a user-defined audio callback function—called 'render'. This render function runs on a dedicated thread and has direct access to the audio buffers used by the hardware ALSA driver (i.e., the ALSA *period*). Similarly to the API of Bela, a 'setup' and a 'cleanup' function are called before the start and after the termination of the render loop. LDSP's simple audio implementation optimizes the use of the phone's resources, enabling advanced audio algorithms and buffer sizes that would typically be prohibitive for Android apps (see next section). Additionally, LDSP can change the mode of operation of the CPU's scaling governor to keep the clock speed at maximum. And as many Android ROMs run preemptive kernels, the framework is designed to try to assign real-time priority to the audio thread, hence further improving timing and performance on supported phones. Figure 1 depicts LDSP audio stack and compares it with that of the modern Android audio app architecture.

4. COMPUTATIONAL PERFORMANCE ANALYSIS

4.1. Software Benchmark

We developed a C++ oscillator bank class to evaluate the computational performance of LDSP. Oscillators are essential components of traditional synthesis techniques and are also used in unconventional DMIs, as demonstrated in previous works (e.g., [23, 24]). While the number of oscillators that can be run in parallel is not a universal metric of an audio application's sonic potential, it provides valuable insight into the musical capabilities of Android phones running LDSP. To further assess the performance gain of LDSP compared to the 'standard' Android audio programming environment, we ran the same code within a custom mobile audio app and quantified the results. Additionally, we benchmarked the oscillator bank on a Bela board for reference.

The oscillator bank is initialized with the number of oscillators and a frequency range when instantiated. The frequencies of the oscillators are linearly spaced within the specified range. Each oscillator consists of a sinusoidal wavetable with linear sample interpolation. When a new sample is requested, the oscillator bank advances all the oscillators, retrieves their samples and sums them into a single value. The total amplitude is normalized to prevent clipping. The source code can be found in the LDSP GitHub repository, under *examples*.

Although the code could be optimized and tailored to individual hardware features, such as multicores or vector floating-point units, we purposely kept the focus on the audio environments and ran identical code on each device. Moreover, our goal was to measure the performance of an application designed by a creative with moderate audio programming skills, as we believe this represents a valuable test case for LDSP, which is designed with accessibility in mind.

When used within LDSP, the oscillator bank is initialized in the setup function and the oscillators' samples are constantly retrieved in the render loop to fill the output audio buffer, following the same code structure used on Bela. In contrast, running the code within an Android app required additional work. We designed a

[7] `https://github.com/victorzappi/LDSP.git`

[8] 'ROM' refers to the combination of firmware and operative system.

[9] `https://github.com/tinyalsa/tinyalsa` [Accessed on 2023/05/26]

simple application following the latest Android app architecture guidelines[10], comprising a minimalistic UI and a small audio engine. The audio engine runs the oscillator bank and sends samples to the phone's audio device, while the UI is used to pass initialization parameters to the oscillator bank, select the audio buffer size and start/stop the audio stream (Figure 2). This app was inspired by the tutorial on wavetable synthesizer by Jan Wilczek[11]. As discussed in Section 2.2, Android app development involves a combination of different programming languages. We built the UI in Kotlin, utilizing the Jetpack Compose framework, which has become a standard for modern Android UI development. For optimal performance, we implemented the audio engine in C++ and used the Oboe audio library, which serves as a wrapper for both the modern AAudio library and the legacy OpenSLES library. Oboe is designed to ensure high performance and provide backward compatibility with older phones and Android versions, which is relevant to our project's aim.

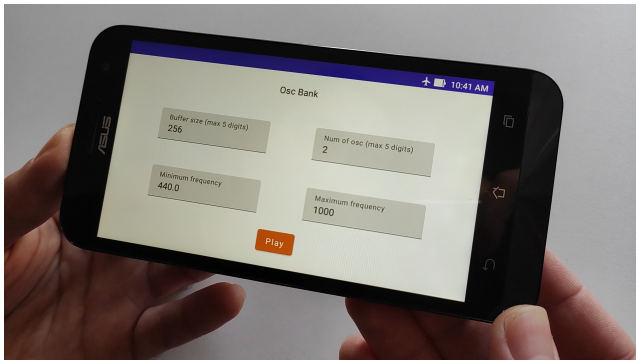

Figure 2: Phone running the oscillator bank Android audio app.

The audio engine includes a setup function that initializes and starts the playback stream, as well as a user-defined callback that is invoked whenever the application's audio buffer needs to be filled. This structure allows the oscillator bank class to be used in a similar fashion as in LDSP. The parameters and functions of the audio engine are mirrored in a Kotlin model, which can be directly accessed from the UI to set the number of oscillators and start/stop playback. The Java Native Interface in Android is used to call the C++ functions directly from the Kotlin model. The source code of the Android application can be found here: `https://github.com/victorzappi/android-osc-bank.git`.

4.2. Methodology

We measured the maximum number of oscillators that a selection of phones could play in parallel using different buffer sizes, using both LDSP and the Android app. In real-time digital audio, increasing the number of samples buffered in memory can improve computational capabilities. However, larger buffer sizes can also increase action-to-sound latency, which tends to limit the usability of the application in interactive scenarios [25]. To obtain a comprehensive view of phone performance, we tested a range of buffer sizes, focusing on powers of two, which is a common set of values for real-time audio applications. Our tests started from the lowest buffer size supported by each phone (typically 32) and increased up to 1024. Each oscillator bank configuration was tested 10 times for a minimum of 45 seconds. If any underruns occurred during this time, the configuration was deemed unreliable for real-time use, and the number of oscillators had to be decreased. We used steps of five oscillators. The first five seconds of each run were discarded from the test window, as considered a warm-up period for the application. All phones were set to airplane mode and no other apps were running during the tests.

It's worth noting that the choice of buffer size is limited by the architecture of Android audio apps. This in turn depends on both the specific hardware and the Android version running on it. For this reason, our tests included also values that depart from the initial pool and comply with each phone's Android Audio HAL as well as the inner workings of the Oboe library. We selected the *low-latency* audio device, when available on each phone, for testing. This is the audio device capable of supporting smaller buffer sizes, and all tests were run using its native sample rate and default number of channels (48 kHz and 2 channels on all tested devices). These settings were retrieved via one of the helper scripts available in LDSP.

In LDSP, we disabled audio capture, sensors, and control inputs/outputs using the appropriate command-line flags. This was done to match the features implemented in the Android app's audio engine and avoid unfair computational overhead. We passed the tested buffer size and the current number of oscillators as command-line arguments to the LDSP executable. As illustrated in Figure 1, the buffer size set within LDSP corresponds with the period size requested from the ALSA driver. We fixed the ALSA ring buffer to two periods.

Within the Android app, we paid particular attention to the audio engine setup and the parameters used to build an efficient audio stream[12]. The engine works with a high-priority callback, and the stream requests exclusive access to the audio device for optimal performance. We also explicitly set the Oboe performance mode to low-latency, which is expected to improve Android's mixer responsiveness.

Figure 1 shows that the audio signal synthesized by Android apps has to pass through several buffering stages before reaching the ALSA driver, namely: the application buffer, which is filled at every Oboe callback; the mixer's buffer, often referred to as the 'internal buffer'; and the Audio HAL buffer, which is filled by the mixer and then passed to the driver. All data transfers between these buffers happen in *bursts*, whose size depends on the audio device. Our app allows requesting the application buffer size from the UI, along with the number of oscillators. Then, the audio engine automatically tries to set the internal buffer to the same size as the application buffer, which is the lowest value possible. In some scenarios, this also allows for samples to be transferred in single bursts (see Section 4.4.1). However, the ALSA period cannot be modified with any Android audio library and is set by the Audio HAL along with the size of the ring buffer, which was fixed to twice the period size on all tested phones. Section 4.4 details how these constraints were taken into account on each phone to assure a fair comparison between LDSP and the app. As a general rule, we started by checking what buffer sizes Oboe managed to set on

[10] `https://developer.android.com/topic/architecture` [Accessed on 2023/05/26]

[11] `https://thewolfsound.com/android-synthesizer-1-app-architecture/` [Accessed on 2023/05/26]

[12] `https://developer.android.com/ndk/guides/audio/aaudio/aaudio#optimizing-performance` [Accessed on 2023/05/26]

each phone, and then we passed to LDSP a single buffer size that combined all Android buffering stages.

4.3. Tested Devices

We tested a total of six phones, which varied in release year, Android version, CPU specifications and overall tier. This heterogeneous pool of devices was selected for two reasons. First, we wanted to assess the performance of LDSP across hardware with different capabilities. Second, we wanted to gather some preliminary data on the actual portability of the environment. This is of particular importance when targeting the upcycling of obsolete devices and was one of the bigger design challenges during the initial stage of development of the project.

The first phone we tested is a Xiaomi Mi 8 Lite. It was released in 2018 and is a high-end device running a stock Android 9 ROM. The second phone is a Huawei P8 Lite from 2015 with Android 5.0 installed. At launch, it was considered a mid-tier device. The third and fourth phones are different models of the Asus ZenFone line. One is a ZenFone 2 Laser (Asus1) released in 2016, equipped with Android 6.0. The other is a ZenFone Go (Asus2) from 2015, running Android 5.1; it was considered a 'budget phone' upon release, but has a slightly higher clock speed than Asus1. The last two phones are both old LG G2 Mini models released in 2014 (LG1 and LG2). LG1 runs Lineage OS 14.1, a custom ROM equivalent to Android 7.0, while LG2 has a stock Android 4.4 ROM. The details and hardware specifications of the tested devices, including Bela's, are listed in Table 1.

Table 1: *Tested devices' details.*

Device	**Year**	**CPU**	**RAM**	**Android**
Xiaomi	2018	octacore 1.8-2.2 GHz	6 GB	9
Huawei	2015	octacore 1.0-1.2 GHz	3 GB	5.0
Asus1	2016	quadcore 1.2 GHz	2 GB	6.0
Asus2	2015	quadcore 1.3 GHz	1 GB	5.1
LG1	2014	quadcore 1.2 GHz	1 GB	7.0
LG2	2014	quadcore 1.2 GHz	1 GB	4.4
Bela	2013	singlecore 1 GHz	512 MB	-

4.4. Results

4.4.1. Android 9

The Xiaomi phone runs Android 9, enabling Oboe to utilize the modern AAudio library. AAudio allows for exclusive access to the audio device, bypassing the internal buffer, whose size was therefor not taken into account when calculating the total buffers tested via LDSP. The app reported a size of 192 samples for both the Audio HAL buffer (i.e., the ALSA period) and the burst size. Therefore, only total buffer sizes larger than 192 samples could be tested on both the app and LDSP. We increased the application buffer size in steps of 192 samples to optimize data transfer to the HAL within the app and avoid unnecessary overhead. The only exception is the starting value of 192 samples, which was approximated by setting a symbolic application buffer size of one sample. Table 2 (left) shows the values we tested, expressed as the combination of Android's application and HAL buffers, as well as the maximum number of oscillators measured in the two environments.

Despite careful choice of audio parameters that could benefit the app, results are largely in favor of LDSP. For sizes that are integer multiples of the HAL buffer, LDSP showed a performance gain that ranged from slightly above 25% to 81%. At 192 samples, the impact of removing the application buffer stage is visible in Android, and the gain reaches almost 700%. Entries below 192, accessible only to LDSP, showcase that the phone is capable of running large numbers of oscillators even with typically small buffer sizes. This reflects the overall high-end specifications of the Xiaomi.

4.4.2. Android 7.0–5.0

The Huawei, the two Asus phones and LG1 all run versions of Android that do not support AAudio[13]. On these devices, Oboe falls back to using the OpenSLES library for audio processing, leading to a series of important limitations on the audio settings and overall performance. Firstly, the audio device cannot be accessed in exclusive mode, meaning that the samples synthesized by the application have to transit through the internal buffer of Android's mixer before reaching the Audio HAL. This was taken into account when computing the total buffer sizes passed to LDSP. Secondly, the buffer size requested by audio apps is ignored as OpenSLES is set to use the most optimal configuration for both the application and internal buffer, as reported in the Audio HAL. The values in use can be checked by inspecting the mixer's status using the command `dumpsys media.audio_flinger` from an Android Debug Bridge shell. Finally, depending on the implementation of the HAL, there is no guarantee that the library matches the actual size of the bursts employed by the audio device, leading to possible overhead during data transfer.

Table 2 (right) displays the results of the tests run on the Huawei. The audio device on this phone only supports a single ALSA period size of 960 samples, as smaller and even larger sizes result in continuous underruns in the audio stream. When running the app, the mixer reported the expected 960 samples for the Audio HAL buffer (matching the supported ALSA period), plus a total of 1924 samples for the application and internal buffers—a surprisingly large value given the overall specifications of the phone. Despite running with a buffer three times smaller, LDSP showcased a performance gain of almost 340%.

Table 3 presents the results for Asus1 (top), Asus2 (middle) and LG1 (bottom). In spite of their lower technical specifications, these phones exhibit good overall audio performance and greater flexibility than the Huawei. They all support Android's *fast mixer*, which enables the use of smaller buffer sizes and requires a lower computational footprint. During app runtime, the fast mixer reported 240 samples for all three buffering stages, resulting in a total size of 720 samples.

On Asus1, OpenSLES effectively matches the optimal buffering configuration and manages to sustain 200 oscillators in real-time. LDSP provides a moderate 22.5% gain at the equivalent buffer size, but with a third of the app's optimal buffer size, it can still run 200 oscillators. Overall, Asus1's audio hardware and firmware are well-suited for interactive audio applications.

The performance of the app on Asus2 falls short compared to other device, with a maximum of only 50 oscillators. This suggests that the configuration employed by OpenSLES is sub-optimal in this case. Nevertheless, LDSP demonstrates the real potential of

[13] AAudio was introduced with Android 8.0 and is not retro-compatible.

Table 2: *Results - Xiaomi Mi 8 Lite (left), Huawei P8 Lite (right) (*native ALSA period size).*

Env. \ Buff.	**1024**	**960**	**768**	**576**	**384**	**192***	**128**	**64**	**32**
LDSP	355	350	335	320	290	235	225	190	85
Android app	-	275	265	255	160	30	-	-	-

Env. \ Buff.	**2884**	**960***
LDSP	-	175
Android app	40	-

Table 3: *Results - Asus ZenFone 2 Laser (top), Asus ZenFone GO (middle), first LG G2 Mini (bottom) (*native ALSA period sizes).*

Env. \ Buff.	**1024**	**720**	**512**	**240***	**128**	**64**	**32**
LDSP	255	245	235	200	180	165	40
Android app	-	200	-	-	-	-	-

LDSP	185	180	175	165	150	135	120
Android app	-	50	-	-	-	-	-

LDSP	180	180	180	175	175	160	110
Android app	-	165	-	-	-	-	-

the phone. When using the same total buffer size as the app, performance improves by 260%. Other buffer sizes yield good numbers of oscillators, albeit lower than those measured on Asus1. Notably, the phone stably runs 120 oscillators at the small buffer size of 32 samples. None of the other tested phones reach this count when using the lowest buffer size supported.

The test results for LG1 reveal good audio capabilities and overshadow the age of the phone. LDSP's performance gain compared to the Android app is at just under 10%. This is likely due to the fact that the CPU has already reached a plateau at a buffer size of 720 samples, where further increase in buffer size does not lead to significant improvements in the maximum count of oscillators. However, LDSP still manages to run a significant number of oscillators at lower buffer sizes, including as low as 32 samples. In fact, at this end of the scale, LG1 outperforms the high-end Xiaomi phone.

4.4.3. Android 4.4 and Bela

LG2 was the last phone to be benchmarked. Unfortunately, we discovered that it is not possible to build the Android app for its outdated Android 4.4 operating system, which is not compatible with the Jetpack Compose framework. Although alternative UI design packages are available for Android versions below 5.0, implementing such a change would have required a massive redesign of the app architecture, as well as a complete re-run of the previous tests. We deemed the app redesign beyond the scope of this work and we decided to only run LDSP on LG2. Table 4 presents the results from LG2 and Bela, both tested using the same buffer sizes, but neither having a direct comparison with the Android app.

LG2's maximum number of oscillators is identical to LG1, except for the value reported at 32 samples. This may be due to the differences in the ROMs loaded on the two phones. Android 4.4 is likely less optimized for real-time audio than Android 7.0; furthermore, LG1 runs a custom ROM, based on Android 7.0 but much more lightweight. Despite these disadvantages, the similar results between the two phones suggest that LDSP's optimizations still manage to harness most of the device's computational power for audio synthesis.

Bela's performance is limited by the BeagleBone Black's lower specifications, but its buffer size scale reaches values that are inaccessible to all the other devices, showcasing its unique ultra-low-latency capabilities.

5. DISCUSSION

LDSP outperforms the Android app in terms of the number of oscillators that can be run on all tested devices and buffer sizes. This suggests that LDSP can better utilize the computational power of the devices for audio synthesis. While this is in line with our expectations due to LDSP's optimized audio stack structure, the degree of improvement is sometimes beyond what we anticipated, even exceeding 100%. As a result, even phones that were previously considered unsuitable for audio applications using standard Android app development, such as ASUS1 and LG2, reveal the actual potential of their underlying hardware when using LDSP. This may open up new musical possibilities for already discarded technology and reminds us that we often underestimate the nature and the origin of the objects we interface with [26, 5]. In fact, one may be surprised to discover that a 2014 Android phone can reliably run more than 150 real-time oscillators using an audio buffer of only 64 samples.

However, when viewed through the lens of sustainability, this seemingly favorable scenario may pose some risks. Like circuit-bent devices [27, 5], upcycled LDSP phones may not age linearly. LDSP's unconstrained access to CPU and hardware capabilities enables the design of audio and musical applications that may put components under significant strain, such as CPU overheating, battery draining and constant high memory data rates. This can lead to a quicker decrease of the phone's lifespan. Nonetheless, we designed LDSP as a tool to repurpose phones that have already reached the end of their product life cycle, at least by modern consumeristic standards. From this perspective, we believe that even reckless phone usage via LDSP would result in an almost neutral environmental impact.

The oscillator bank experiment shows how LDSP empowers developers to optimize the balance between performance and responsiveness of their applications, by fine-tuning buffer size. While this may seem like a minor detail to experienced audio programmers and creatives, our tests expose the limitations of Android in this regard. Modern Android versions offer little flexibility when it comes to adjusting the overall buffering mechanism, while older versions such as Android 7.0 and lower straight remove this possibility.

Our experiments with various Android devices have helped us understand the rationale behind these architectural constraints. For instance, some devices like Huawei have limited audio codecs that support fixed periods only. To overcome this, Android relies on a multi-stage buffering mechanism that sits atop the Audio HAL and low-level audio driver, granting applications enough time to complete audio synthesis or processing even when the native period

Table 4: *Results - second LG G2 Mini (LG2) and Bela.*

Buff. / Env.	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**
LG2	180	180	180	175	160	100	-	-	-	-
Bela	100	105	105	80	80	80	80	75	70	25

may not be enough. Conversely, LDSP buffers are always constrained by the native period size of the audio device. While the Android multi-buffering mechanism is useful for general-purpose multimodal applications, it adds unnecessary overhead and can introduce buffers whose sizes are inappropriate for responsive audio applications, as our results suggest.

Portability is an important feature that emerges from LDSP's results. We could have possibly either upgraded the ROM or downgraded the UI package to run the Android app on LG2. Yet, the fact that LDSP seamlessly runs on all phones including this very old one is a way more valuable result. The presence of ALSA low-level drivers is the only strict requisite for LDSP to be supported on a phone. ALSA started being included in the Android kernel since Android 2.3 and it is now the most widely used audio driver across all brands and models of phones. This means that phones released as early as 2010 are very likely to support LDSP and run the same that code we tested on our 2018 Xiaomi. While harder to find and more modest in terms of hardware as well as software capabilities, such old phones can still be spotted in flee markets, garages and even in secluded drawers within our very homes. We believe they can be instrumental to unleashing creativity in spite of and because of their limitations [23], and we are looking forward to testing one.

Compared to standard Android development, LDSP offers a streamlined solution that eliminates the need for developers to learn and use different packages and frameworks based on the phone's age and setup. Instead, LDSP is based on standard C++, making it a one-size-fits-all solution that also requires less hardware and software for development. Whereas Android Studio is typically the only option for deploying an application on a phone, LDSP is development environment-agnostic and allows for the use of leaner editors, resulting in faster and less memory/power-intensive compilation.

Additionally, the comparison with Bela showcases how LDSP offers a low entry fee for creatives. Bela can leverage block-based processing on the onboard NEON vector floating point unit to reach 700 oscillators [23]. When combining C++ and Assembly, these results hold for buffers as small as 16 samples. While similar optimization techniques can be carried out on phones via LDSP[14], a person who wants to repurpose old technologies may not be familiar with the hardware specifics of a discarded phone, nor may they want to delve into low-level optimization audio practices. Nonetheless, our entry-level code yielded better results than Bela's even on budget/old phones, suggesting a larger audio application domain with minimal coding effort.

6. CONCLUSIONS

In this paper, we discussed how LDSP can be used to harness the potential of old Android phones and foster the design of creative audio applications. We ran an experiment using six different Android phones to test the performance of an oscillator bank application built using the LDSP C++ framework. Results suggest that LDSP outperforms the standard Android app in terms of the number of oscillators that can be run on all tested devices and buffer sizes. This is likely due to LDSP's optimized audio stack structure, which better utilizes the computational power of the devices for audio synthesis.

While often referring to the impact that the size of the buffer has on the responsiveness of the application, more tests are necessary to measure the actual latency of audio applications designed with LDSP and compare them with results obtained with equivalent Android applications.

In our judgement, the central emphasis placed by LDSP as a project on upcycling and reclaiming conventional technology invites innovative approaches to engage with it, both practically and politically. This engagement entails acknowledging our agency and responsibility towards the technologies we have created, used and discarded. By reusing and exploring new ways to interact with off-the-shelf devices, we shift our focus towards sustaining them in a manner that nurtures creativity rather than solely pursuing the allure of the latest technology [28].

The utilization of ready-made technologies also holds the potential for maintenance through community-driven practices and shared knowledge [29]. Given the abundance of Android technology expertise available in varying degrees, the likelihood of continued support is somewhat assured. Therefore, it becomes paramount for us to collaborate with musicians and developers, as such partnerships would expand the project's capabilities to cater to diverse needs, interests, and skill sets while fostering a sense of community across different spheres of action.

LDSP was designed with accessibility in mind, as evident through its collection of examples and libraries, as well as its overall simplicity. However, it does require basic C++ or equivalent coding skills to fully explore its potential. In this study, we highlighted the advantages of working with low-level C++ for achieving optimal audio performance. However, it is important to note that this comes at the cost of a less accessible environment compared to other Android audio frameworks. To address this issue, we have recently introduced support for Pure Data patches by integrating libpd directly into the core low-level audio engine of LDSP. This enhancement offers users the flexibility to build their LDSP applications using Pure Data exclusively or to combine Pure Data with C++, allowing for a tailored balance between code complexity and performance. In fact, it is worth mentioning that the use of libpd introduces some computational overhead that may impact the performance of audio applications (this is seen in Bela too). We plan to conduct further tests to quantify this impact within the LDSP environment.

[14] All the tested devices come equipped with the NEON.

7. REFERENCES

[1] Alessio Balsini, Tommaso Cucinotta, Luca Abeni, Joel Fernandes, Phil Burk, Patrick Bellasi, and Morten Rasmussen,

"Energy-efficient low-latency audio on android," *Journal of Systems and Software*, vol. 152, pp. 182–195, 2019.

[2] Margaret Butler, "Android: Changing the mobile landscape," *IEEE Pervasive Computing*, vol. 10, no. 1, pp. 4–7, 2011.

[3] Andrew McPherson, "Bela: An embedded platform for low-latency feedback control of sound," *The Journal of the Acoustical Society of America*, vol. 141, no. 5, pp. 3618–3618, 2017.

[4] Zak Argabrite, Jim Murphy, Sally Jane Norman, and Dale Carnegie, "Technology is Land: Strategies towards decolonisation of technology in artmaking," in *NIME 2022*, 2022.

[5] Enrico Dorigatti and Raul Masu, "Circuit bending and environmental sustainability: Current situation and steps forward," in *NIME 2022*, 2022.

[6] Joe Cantrell, "The timbre of trash: Rejecting obsolescence through collaborative new materialist sound production," *JAAAS: Journal of the Austrian Association for American Studies*, vol. 1, no. 2, pp. 217–229, 2020.

[7] Vítor N Palmeira, Graziela F Guarda, Luiz Fernando Whitaker Kitajima, et al., "Illegal international trade of e-waste–europe," *Detritus*, vol. 1, no. 1, pp. 48, 2018.

[8] Marina Proske, Janis Winzer, Max Marwede, Nils F Nissen, and Klaus-Dieter Lang, "Obsolescence of electronics-the example of smartphones," in *2016 Electronics Goes Green 2016+(EGG)*. IEEE, 2016, pp. 1–8.

[9] Parvez Alam, "Hacking into e-waste," *Material World*, vol. 28, no. 7/8, pp. 24–27, 2020.

[10] João Tragtenberg, Gabriel Albuquerque, and Filipe Calegario, "Nime 2021," in *Gambiarra and Techno-Vernacular Creativity in NIME Research*, 2021.

[11] Rômulo Vieira and Flávio Schiavoni, "Fliperama: An affordable arduino based midi controller," in *NIME 2020*, 2020.

[12] Ignacio Orobitg, Adolfo Subieta, Federico Uslenghi, and Federico Wiman, "El desarrollo de la música electroacústica en buenos aires," *Revista del Instituto de Investigacion Musicologica Carlos Vega*, 2003.

[13] Laewoo Kang, "Echo(): Listening to the reflection of obsolete technology," *Proceedings of the 2017 ACM Conference Companion Publication on Designing Interactive Systems*, 2017.

[14] JGA Lima, "The gatorra: a technologically disobedient instrument for protest music," in *xCoAx 2018: Proceedings of the Sixth Conference on Computation, Communication, Aesthetics & X*, 2018, pp. 55–64.

[15] Caleb Stuart, "Damaged sound: Glitching and skipping compact discs in the audio of yasunao tone, nicolas collins and oval," *Leonardo Music Journal*, vol. 13, no. 1, pp. 47–52, 2003.

[16] Timothy Tate, "The Concentric Sampler: A musical instrument from a repurposed floppy disk drive," in *NIME 2022*, 2022.

[17] Karl Yerkes, Greg Shear, and Matthew James Wright, "Disky: a diy rotational interface with inherent dynamics," in *NIME 2010*, 2010.

[18] Benjamin Taylor, Jesse T Allison, William Conlin, Yemin Oh, and Daniel Holmes, "Simplified expressive mobile development with nexusui, nexusup, and nexusdrop.," in *NIME 2014*, 2014, pp. 257–262.

[19] Daniel Iglesia and Iglesia Intermedia, "The mobility is the message: The development and uses of mobmuplat," in *Pure Data Conference (PdCon16). New York*, 2016.

[20] Romain Michon, Yann Orlarey, Stéphane Letz, Dominique Fober, and Catinca Dumitrascu, "Mobile music with the faust programming language," in *Perception, Representations, Image, Sound, Music: 14th International Symposium, CMMR 2019, Marseille, France, October 14–18, 2019, Revised Selected Papers*, 2021, pp. 307–318.

[21] Ben Cahill and Stefania Serafin, "Guitar simulator: An audio-haptic instrument for android smartphones.," in *The Seventh International Workshop on Haptic and Audio Interaction Design August 23-24 2012 Lund, Sweden*, 2012, pp. 19–20.

[22] Carla Sophie Tapparo, Brooke Chalmers, and Victor Zappi, "Leveraging android phones to democratize low-level audio programming.," in *NIME*, 2023.

[23] Victor Zappi and Andrew McPherson, "Design and use of a hackable digital instrument," in *Proceedings of the International Conference on Live Interfaces*, 2014.

[24] Adnan Marquez-Borbon, "Perceptual learning and the emergence of performer-instrument interactions with digital music systems," in *Proceedings of A Body of Knowledge - Embodied Cognition and the Arts Conference 2016*, 2016.

[25] Andrew P McPherson, Robert H Jack, Giulio Moro, et al., "Action-sound latency: Are our tools fast enough?," in *NIME 2016*, 2016.

[26] Alexandra Rieger and Spencer Topel, "Driftwood: Redefining sound sculpture controllers.," in *NIME 2016*, 2016, pp. 158–159.

[27] M Premalatha, Tabassum-Abbasi, Tasneem Abbasi, and SA Abbasi, "The generation, impact, and management of e-waste: State of the art," *Critical Reviews in Environmental Science and Technology*, vol. 44, no. 14, pp. 1577–1678, 2014.

[28] Steven J Jackson, "11 rethinking repair," *Media technologies: Essays on communication, materiality, and society*, pp. 221–39, 2014.

[29] Mela Bettega, Raul Masu, Nicolai Brodersen Hansen, and Maurizio Teli, "Off-the-shelf digital tools as a resource to nurture the commons," in *Proceedings of the Participatory Design Conference 2022-Volume 1*, 2022, pp. 133–146.

[30] Ana Delgado and Blanca Callén, "Do-it-yourself biology and electronic waste hacking: A politics of demonstration in precarious times," *Public Understanding of Science*, vol. 26, no. 2, pp. 179–194, 2017.

[31] Juan Pablo Martinez Avila, João Tragtenberg, Filipe Calegario, Ximena Alarcon, Laddy Patricia Cadavid Hinojosa, Isabela Corintha, Teodoro Dannemann, Javier Jaimovich, Adnan Marquez-Borbon, Martin Matus Lerner, Miguel Ortiz, Juan Ramos, and Hugo Solís García, "Being (A)part of NIME: Embracing Latin American Perspectives," in *NIME 2022*, 2022.