Energy-Efficient Low-latency Audio on Android

Alessio Balsini[a,b], Tommaso Cucinotta[a], Luca Abeni[a], Joel Fernandes[b], Phil Burk[b], Patrick Bellasi[c], Morten Rasmussen[c]

[a] *Scuola Superiore Sant'Anna, Via Moruzzi, 1, 56124 Pisa, Italy*
[b] *Google, Googleplex, 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA*
[c] *Arm Limited, 110 Fulbourn Rd, Cambridge CB1 9NJ, UK*

Abstract

Counting more than two billion devices, Android is nowadays one of the most popular open-source general-purpose operating systems, based on Linux. Because of the diversity of applications that can be installed, it manages a number of different workloads, many of them requiring performance/QoS guarantees. When running audio processing applications, the user would like an uninterrupted, glitch-free, output stream that reacts to the user input, typically with a delay not bigger than $4 - 10$ ms, while keeping the energy consumption of the mobile device as low as possible.

This work focuses on improvements to the real-time audio processing performance on Android that preserves energy-efficiency. Such improvements are achieved by using a deadline based scheduler and an adaptive scheduling strategy that dynamically and proactively modulates the allocated runtime. The proposed strategy is evaluated through an extensive experimentation, showing that 1) compared to the existing way to ensure low-latency audio processing, the proposed mechanism provides an energy saving of almost 40%, and 2) compared to the existing way to achieve a good balance between power consumption and latency in a glitch-free audio processing experience, the proposed solution reduces audio latency from 26.67 ms to 2.67 ms, at the expense of a limited

Email addresses: alessio.balsini@santannapisa.it (Alessio Balsini), tommaso.cucinotta@santannapisa.it (Tommaso Cucinotta), luca.abeni@santannapisa.it (Luca Abeni), joelaf@google.com (Joel Fernandes), philburk@google.com (Phil Burk), patrick.bellasi@arm.com (Patrick Bellasi), morten.rasmussen@arm.com (Morten Rasmussen)

power consumption increase of 6.25%.

Keywords: android, Linux kernel, real-time scheduling, adaptive reservations, energy efficiency, low-latency audio

1. Introduction

A significant limitation of the Android[1] operating system (OS) has always been the difficulty in providing low-latency audio features[2], which are often required in a significant number of interactive multimedia applications, like professional grade multimedia. An audio processing delay in the range of $4-10\ ms$ is well-known among musicians and digital sound practitioners as the maximum acceptable one for professional audio. Indeed, the guidelines for Android low-latency application developers[3] suggest a period of $2-5\ ms$ for the audio processing pipeline, so as to keep the overall audio production latency sufficiently below the 30 ms threshold that is equally well-known to be perceivable as echo by the human ear [1, 2].

One of the obstacles in achieving low-latency on Android platforms is the heterogeneity of devices running Android, having different hardware capabilities, as well as the lack of proper interaction models between applications and kernel, thus requiring several software abstraction layers and big audio buffers, both resulting in the capability of generating a smooth audio output, at the price of an audio latency increase.

Multimedia applications, as shown in Figure 1, can be realized on Android with many different APIs. From the Java language, a number of classes in the `android.media` package can be conveniently used for playing or recording audio

[1] This work refers to the *master* branch of the Android Open Source Project (AOSP) synchronized on the 21st of May 2018 at 4:36pm CET, an under-development version of Android Q.

[2] Audio latency estimations can be measured with the Superpowered Mobile Audio Latency Test App, and the results are collected and shown at the following address: `https://superpowered.com/latency`.

[3] More information at: `https://source.android.com/devices/audio/latency/design`.

locally, managing audio files, or streaming audio from a remote source. However, this option forces the use of large buffers, causing non-negligible audio latency. For interactive, low-latency scenarios, applications need to possess a C/C++ component that uses the available low-latency C/C++ APIs, e.g., `OpenSL ES` or `AAudio` (more details will be provided later in Section 4).

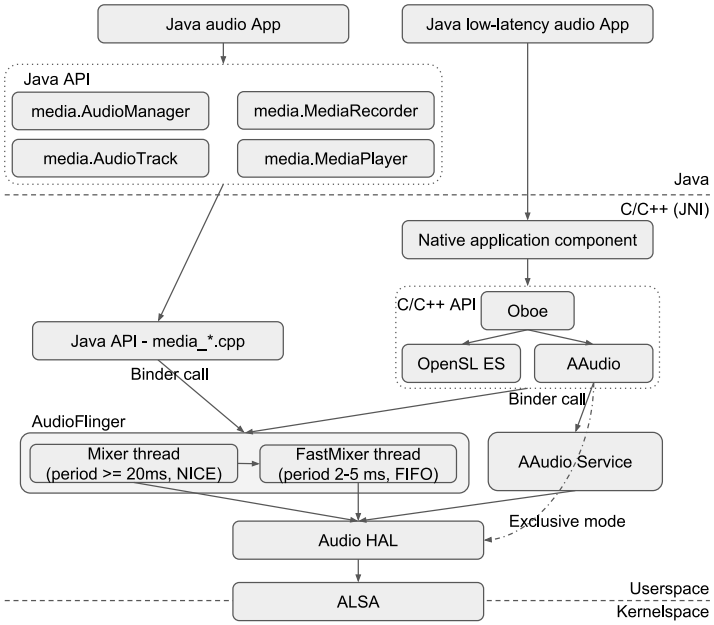


Figure 1: Overview of the Android audio architecture.

When using the low-latency audio APIs, a real-time thread using the `SCHED_FIFO` scheduling discipline is used, which, along with `SCHED_RR`, is part of the `SCHED_RT` scheduling class of the Linux kernel. Because of the energy requirements of the target mobile devices, `SCHED_RT` has been integrated with the `schedutil` policy of the `CPUFreq`[4] framework that, according to internal heuristics, estimates the expected overall system workload and sets the CPU frequency.

Unfortunately, this approach slowly reacts to dynamic workload changes, such as sudden spikes up of CPU demand due to, e.g., a sudden increase in

[4]More information at: `https://www.kernel.org/doc/Documentation/cpu-freq/`.

the number of voices being synthesized in software. Indeed, by the time the `schedutil` heuristic realizes that the CPU workload increased, and a frequency increase is needed, it might already be too late, and the audio pipeline exhibits an audible glitch. This problem can be mitigated by increasing the audio buffer size, resulting in a larger latency of the audio processing pipeline, so that the system has more time to adapt the CPU frequency before an audio glitch happens, which ultimately makes the current solution not suitable for professional audio applications. Alternatively, low latency can be achieved by locking the CPU to the maximum frequency when a `SCHED_RT` task is present, as done in the mainline Linux kernel (more details in Section 4.3), leading to an unacceptable increased power consumption for mobile devices.

This work proposes a novel, exploratory solution to tackle this problem, based on: 1) adapting the `AAudio` framework so as to switch to a deadline-based programming paradigm which uses the `SCHED_DEADLINE` scheduling class recently added to the Linux kernel [3]; this, differently from `SCHED_RT`, allows `schedutil` to adapt the frequency coherently with the real-time workload as known to `SCHED_DEADLINE`, preserving the timing constraints of the application; and 2) by extending the `AAudio` API by introducing a new mechanism to notify the system about workload demand changes, which are used for a proactive update of the computational bandwidth, thus of the CPU frequency. Experimental results conducted over an Android board demonstrate the advantages of the proposed approach in both application responsiveness and energy efficiency.

1.1. Paper Structure

This paper is structured as follows. Section 2 overviews prior literature on the topic, focusing on research works proposing adaptive reservation-based and energy-efficient schedulers for real-time workloads, and solutions specifically designed for Android. Section 3 recalls a few important concepts about CPU scheduling for soft real-time and multimedia workloads in General-Purpose Operating System (GPOS) kernels. Section 4 summarizes common background concepts on the Android audio architecture, power management features within

the Linux and Android kernels and the recently available deadline-based scheduling in Linux. Section 5 describes our proposed technique for dealing with low-latency audio applications in an energy-efficient way, provides details about the adopted feedback-based and proactive adaptation strategies, and presents details of how the proposed method has been evaluated using the SynthMark benchmarking framework. Section 6 describes the experimental results obtained with our implementation of the proposed technique on a real platform running Android. Finally, conclusions are drawn in Section 7 and some ideas for further research and experimentation on the topic are proposed in Section 8.

2. Related Work

Several prior efforts exist in the research literature addressing the problem of ensuring predictable execution and performance stability of multimedia applications and soft real-time workloads on general-purpose operating systems, investigating on possible trade-offs among achieved performance and power consumption on mobile devices.

In this paper we propose to adopt a proactive resource adaptation strategy, combining it in a novel way with various techniques from literature: real-time scheduling (and reservation-based scheduling [4] in particular, see Section 3.2 for details); feedback-based adaptation of the scheduling parameters; and Dynamic Voltage and Frequency Scaling (DVFS). In this section, we present an overview of the major related research works, which will be compared in Section 2.4 according to a few criteria, including their suitability for hard real-time or soft real-time and multimedia, their inclusion of adaptive strategies, their support for multi-processor platforms and their attention to energy efficiency.

2.1. Adaptive Resource Reservations

One of the most important problems encontered when trying to use real-time scheduling (and reservation-based scheduling in particular) in a General-Purpose Operating System (GPOS) is the difficulty in properly tuning the (real-time) scheduling parameters. For example, it is not easy to estimate the needed

resource allocation, given the plethora of hardware platforms each application can be deployed onto. This urges developers into enriching their applications with the logic that is needed to profile the time-critical tasks over the specific hardware they are deployed onto, so to be able to communicate the correct computational requirements to the kernel.

As a consequence, some form of feedback mechanism can be added to a real-time scheduler to properly adapt its behavior, as proposed by Stankovic et al. [5] and Lu et al. [6]. Focusing on reservation-based scheduling, *adaptive reservation* techniques have been extensively studied, which are based on theoretical arguments from control theory to analyze the closed-loop performance, as done for example by Abeni et al. [7]. In particular, Stankovic et al. [5] and Lu et al. [6] proposed to use the feedback loop to adjust the system load so that the amount of missed deadlines is kept under control. On the other hand, Abeni et al. [7] proposed an underlying reservation-based scheduling discipline, and used a feedback loop to dynamically adapt the reserved runtime, so as to track the dynamic workload of the real-time application, using a linearized controller whose stability has been investigated recurring to control-theoretical arguments.

Adaptive reservations have been extended in a variety of ways: Cucinotta et al. [8] and Abeni et al. [9] proposed to use non-linear controllers, as well as to provide probabilistic real-time guarantees [10]; Palopoli and Cucinotta [11] extended the approach further to support pipelines of tasks with end-to-end deadline guarantees. Implementations based on Linux have also been released as open-source [12]. Moreover, Cucinotta et al. [13] proposed also to integrate adaptive reservations with application-level adaptation, and with CPU reclaiming mechanisms [14].

Concerning the real-time performance of audio applications on Linux, experimentation with deadline-based real-time scheduling has been previously conducted by Cucinotta et al. [15] using the JACK[5] low-latency audio infrastructure. JACK was modified to use AQuoSA [12], an old single-processor im-

[5]More information at: `http://jackaudio.org`.

plementation on Linux of the Constant Bandwidth Server (CBS) by Abeni et al. [16], an algorithm based on EDF scheduling; however, power management and multi-core scheduling were not considered in that study, and only history-based runtime adaptation was investigated.

2.2. Energy-Aware Real-Time Scheduling

There is a large amount of literature on energy-aware scheduling algorithms, both for real-time and general purpose systems. Useful overviews on the topic have been published for example by Stangaciu et al. [17] and Bambagini et al. [18].

When real-time performance is not important, an estimation of the workload based on a moving average can be used to drive the frequency scaling mechanism, as shown by Hu et al. [19]. On the other hand, when respecting deadlines is important, as in this paper, real-time schedulability analysis can be used to reduce the energy consumption without breaking (hard or soft) real-time guarantees. For example, Pillai and Shin [20] proposed to set the CPU frequency based on the real-time tasks' utilization, and then dynamically exploit the slack time to further decrease the CPU frequency. This technique only works for periodic tasks. Similar algorithms, also based on the periodic task model, have been used by Aydin et al. [21], who proposed more advanced algorithms for exploiting the slack time, that result in improved power saving.

Zhu and Mueller [22] proposed a different approach, that dynamically scales the CPU frequency based on a feedback mechanism. In particular, this approach is based on splitting each job of a task in two parts, optimistically executing the first part at a low frequency and then increasing the CPU frequency when the second part executes. The feedback mechanism, based on the well-known proportional-integral-derivative (PID) controller, is used to adapt the size of the first part of a job to match its execution time.

As an alternative, it is possible to characterize the CPU requirements of a taskset through a *demand bound function* (representing the worst-case CPU requirements of the taskset in a time interval), and to compute the frequency

requirements of the taskset by comparing the demand bound function with the amount of execution time supplied by a CPU running at a given frequency, as shown by Kim et al. [23]. This approach is based on using the so-called compositional scheduling framework analysis introduced by Shin and Lee [24].

Some previous works also investigated implementation issues (instead of only focusing on the theory behind the scheduling/DVFS policies) by considering, for example, the impact of the used frequency scaling mechanism, as studied by Zhu and Mueller [25].

Excluding some notable exceptions like the one by Almoosa et al. [26], most of the previous work only considered single-core / single-CPU systems.

Several authors also have considered energy-aware adaptive scheduling of soft real-time and multimedia applications, for example using frequency scaling to deliver probabilistic real-time guarantees, as done by Zha et al. [27]: instead of scaling the CPU frequency so that every task is guaranteed to respect all of its deadlines (as, for example in the previously mentioned works [20, 21]), the frequency is set so that the probability to respect a deadline is controlled. Another approach for energy saving in soft real-time systems is presented by Lorch and Smith [28], where it is shown how to set the CPU frequency after a deadline has been missed (post-deadline speed schedule) so that some utility functions are optimized. Pouwelse et al. [29, 30] presented some experiments with dynamic frequency scaling (driven by a user-space program that scales the CPU frequency so that the CPU load is about constant) applied to a video decoding application (an H.263 decoder). Kumar et al. [31] proposed a prediction mechanism for fixed-priority scheduling of soft real-time periodic tasks. However, these techniques are based on heuristics and cannot provide guarantees to hard real-time tasks. Qadi et al. [32] presented the DVSST algorithm that reclaims the unused bandwidth of sporadic hard real-time tasks.

Yuan et al. proposed GraceOS [33], which uses a monitoring mechanism to adapt the scheduling parameters of the applications and frequency switching of the platform so that quality of service requirements and energy consumption constraints are respected. Cucinotta et al. [13] used an adaptive multi-layer

control architecture to achieve similar objectives, where multiple levels of adaptation (application level, middleware level and scheduling level) were used to switch the application mode of operation, the CPU frequency, and the scheduling parameters in order to optimize quality of service and energy consumption.

An alternative approach still proposed by Cucinotta et al. [34] was using a userpace daemon (a modified `powernowd` daemon) to adapt the CPU frequency while not switching to power modes that would break schedulability of a system that has already accepted real-time applications.

A different view on power management is offered by the Q-RAM framework by Rajkumar et al. [35, 36], which allows one to model resource allocation in real-time systems as an optimization problem, allocating various kinds of resources (memory, CPU, network bandwidth) to tasks so that some cost functions are minimized (or utility functions are maximized). Using Q-RAM, power management can be modeled by considering the consumed power as a cost.

Some of the previously mentioned works rely on feedback-based adaptation, where the resource allocation is changed based on observed past history of the system. In this paper, we are proposing a new proactive approach instead (see Section 5), where a novel API is introduced to notify the kernel about an upcoming change in workload demand, and ask for the needed resource allocation increases *ahead of time*. The resulting adaptive and proactive reservation-based strategy is built on top of a DVFS algorithm inspired by GRUB-PA by Scordino et al. [37]. GRUB-PA is a generic algorithm supporting periodic, sporadic and aperiodic tasks, and allowing to provide both soft and hard real-time guarantees. In more details, GRUB-PA works by scaling the CPU frequency based on the *active utilization* or real-time tasks, as tracked by the GRUB scheduling algorithm (informally speaking, the active utilization of real-time tasks can be seen as the fraction of CPU time used by such tasks).

The original GRUB and GRUB-PA algorithms were uni-processor only, but the reclaiming and bandwidth accounting mechanisms have been extended by Abeni et al. to support multiple CPUs/cores [38].

Since it can be integrated in systems containing non real-time tasks too,

GRUB-PA has been the perfect candidate for supporting frequency scaling in operating systems like Linux, and one of its variations has been implemented in the `schedutil` frequency scaling governor, as described by Scordino et al. [39, 40]. In practice, the GRUB's active utilization is used by `schedutil`, along with the load estimates available for the other scheduling classes, to scale the frequency of each CPU core so that all of the hosted tasks can be accommodated as due to their demand.

2.3. Android

Focusing on Android, prior research literature also exists, addressing real-time issues, priority inversion and other performance-oriented aspects of the OS. As highlighted by Levin [41], a form of priority inversion mitigation has been integrated since a long time within the Binder IPC framework, extensively used throughout Android applications and services. The modifications allow for preserving, across synchronous remote procedure calls (RPCs), the `nice` level of the calling thread. More recently, also the real-time priority of real-time tasks is preserved, as shown by Kalkov et al. [42][6].

Further proposed modifications for enhanced real-time support in Android include the works by Kalkov et al. [43] and Yan et al. [44, 45], adopting a real-time Java Virtual Machine run-time platform to control interferences due to the garbage collector, enhancing the memory allocator, and improving the accessibility of scheduling services from unprivileged applications, without the need for a Binder call to a privileged process. Besides these, an additional work by Yan et al. [46] proposed to extend the Android interfaces for the development of soft real-time applications, by introducing statically specified memory bounds and priority awareness.

[6] For details, refer to commit history of `binder.c` as available at: `https://android.googlesource.com/kernel/common/+/android-4.9/drivers/android/binder.c`.

2.4. Discussion

The major existing prior literature on adaptive real-time, power-aware scheduling for multimedia and soft real-time applications, which has been reviewed above, can be categorized as in Table 1, where different features are highlighted for each of the work, including: support for soft or hard real-time deadlines; support for feedback-based adaptation of the scheduling parameters or proactive adaptation; whether some form of closed-loop analysis is provided (either deterministic or probabilistic); whether the work is validated through a real implementation of the technique, or its simulation, or by theoretical arguments; whether the focus is on multimedia use-cases or others; whether the framework supported only single- or also multi-processor scheduling; and finally whether energy-efficiency has been addressed. The last line in the table refers to the mechanism being proposed in this work. Moreover, the works emphasized in italics are other works that have been combined with the feed-forward mechanism in the approach proposed in this paper.

The present work is the first one focusing on the Android operating system, combining the `SCHED_DEADLINE` real-time scheduling policy of the Linux kernel with the `CPUFreq` power management subsystem and its `schedutil` governor, considering an underlying heterogeneous processing architecture with CPUs having different processing capacities (e.g., Arm `big.LITTLE`), and *introducing a proactive adaptation approach*. This lets applications declare in advance their expected workload changes, so that *scheduling parameters adaptations can be anticipated*, resulting in an effective capability to keep a low-latency glitch-free playback in the presence of heavy fluctuations of the processing demand, while at the same time limiting power consumption to the minimum. Effectiveness of the presented technique will be quantified through the experimental results presented in Section 6, where the focus will be on comparing the proposed approach with what is currently available on nowadays Android platforms. An extensive experimental comparison with one or more of the approaches on adaptive reservation-based and energy-aware real-time scheduling strategies reviewed above, is something we leave out of this paper, as possible additional future

Table 1: Comparative chart of prior literature. Deadlines support is one of **H**ard, **S**oft, **N**one. Validation is one of **L**inux, **A**ndroid, **S**imulation, **T**heoretic. Use-case is one of **Mu**ltimedia, **Of**fice, **Sy**nthetic, **Av**ionics, **Mi**litary, **Me**dical.

Work	Deadlines Support	Feedback Scheduling	Proactive Scheduling	Closed-Loop Analysis	Validation	Use-Case(s)	Multi-Processor	Energy-Efficiency
Hu et al. [19]	N	No	No	—	L	Mu	No	Yes
Rajkumar et al. [4]	H/S	No	No	—	L	Mu	No	No
Rajkumar et al. [35, 36]	H/S	No	No	—	L	Mu	Yes	Yes
Pillai et al. [20]	H	No	No	—	S/L	Sy	No	Yes
Aydin et al. [21]	H	No	No	—	S	Sy	No	Yes
Kim et al. [23]	H	No	No	—	T	Av	No	Yes
Lorch et al. [28]	S	No	No	—	S	Of, Mu	No	Yes
Pouwelse et al. [29, 30]	S	No	No	—	L	Sy, Mu	No	Yes
Kumar et al. [31]	S	No	No	—	S	Av, Sy	No	Yes
Zhu et al. [22]	H	Yes	No	Yes	S/L	Sy	No	Yes
Almoosa et al. [26]	N	Yes	No	Yes	S	Sy	Yes	Yes
Yuan et al. [33]	S	Yes	No	Yes	L	Mu	No	Yes
Kalkov et al. [42, 43]	S	No	No	—	A	Sy	No	Yes
Yan et al. [44, 45, 46]	H/S	No	No	—	A	Mi, Me	Yes	No
Cucinotta et al. [14, 12, 10, 13, 8, 34], Abeni et al. [7]	S	Yes	No	Yes	L	Mu	No	Yes in [13, 34]
Cucinotta et al. [15]	S	Yes	No	No	L	Mu	No	No
Scordino et al. [39, 40]	S	No	No	—	L	Sy	Yes	Yes
This work	**S**	**Yes**	**Yes**	**No**	**A**	**Mu**	**Yes**	**Yes**

work that would validate better our achieved results. A remarkable observation, though, is the one that, without the proactive notification mechanism that is introduced in this paper, any feedback-based methodology will not be able to react to workload changes and anticipate power adjustments as quickly and on-time, as in the mechanism proposed herein.

Albeit the approach we are presenting in this paper builds upon the use of a number of mechanisms (mostly ours) which have already been presented in the past, their combination with a proactive feedback approach on Android heterogeneous platforms described in Section 5, the implementation and the experimental evaluation carried out in Section 6, are all novel elements presented here for the first time. These mark one undoubtedly useful step towards the design and engineering of novel mechanisms in the context of soft real-time multimedia for Android, where the use of even widely studied techniques from the theory of real-time systems is all but straightforward.

3. Background on Scheduling for Soft Real-Time Applications

To support low-latency audio applications in Android, this work combines proactive adaptation with some well-known real-time scheduling techniques that are already implemented in some OS kernels (and in the Linux kernel in particular). In this section, we briefly recall the theory and practice on which this work relies, starting from CPU scheduling and real-time scheduling.

3.1. GPOS Process Schedulers

GPOSes have been incorporating for decades scheduling mechanisms based on priorities, and APIs allowing for raising or lowering a process priority, using for example a real-time scheduling discipline or the `nice`-ness of processes on OSes conforming to the POSIX [47] standard. However, GPOSes have also traditionally been including a variety of heuristics for handling multimedia applications in the context of desktop systems, specifically for letting multimedia applications coexist with CPU-intensive processing workloads while keeping a

smooth and glitch-free playback. These heuristics have been based on the concept of automatically distinguishing *interactive* workloads, which are usually characterized by alternating sequences of short-lived processing and sleeping time frames, from *batch* workloads, typically having a much longer duration of CPU-intensive activities. Then, the detected interactive processes are transparently boosted in their priorities with respect to batch ones, without requiring the developers of multimedia applications to make any specific adjustment. For example, the Linux `SCHED_OTHER` policy has been using a heuristic of this kind [48], tracking per-task sleep vs. ready-to-run time windows, and boosting the dynamic priority of a task after wake-up, while de-boosting it while running continuously. This allows us to run long-running, CPU-intensive applications such as compilation of complex software suites, number-crunching or CAD (computer-aided design) computations, while using interactive desktop applications, such as audio/video players, web browsers, e-mail clients, with a very good responsiveness.

Nevertheless, these heuristics are known to result in quite an unstable allocation of the CPU to multimedia applications. So, these have been traditionally designed with large pre-computed buffers, or with the ability of application-level adaptation to the available resources, as typically done in video applications that skip frames when needed. However, in the case of low-latency requirements, it is common to resort to APIs for either increasing the `nice` level into a general-purpose scheduler or switching to a predictable and deterministic real-time scheduling policy, typically available as the POSIX `SCHED_FIFO` or `SCHED_RR` disciplines [47]. By using these mechanisms, developers of multimedia applications may explicitly raise the priorities of the time-critical tasks in their applications, so that whenever they are ready to run they will preempt other lower-priority tasks, keeping the desired interactivity level.

Unfortunately, these techniques are known to work well only in typical setups where users run just one main application. Whenever we start dealing with a multitude of (soft) real-time tasks, it becomes cumbersome to tune their priorities properly, and more sophisticated techniques are needed, for ensuring

a smooth component-based approach to the design and deployment of complex software.

3.2. Reservation-Based Scheduling

The research literature on real-time systems includes many approaches focusing on CPU scheduling and the application of *reservation-based scheduling* techniques [4]. These let the OS kernel guarantee a timely allocation of the CPU to competing applications according to their individual requirements, including their specific time granularity. The traditional priority-based method of dealing with multiple heterogeneous real-time activities requires knowledge of the whole set of activities in the system, in order to sort priorities properly, e.g., using the well-known *rate-monotonic* assignment [49]. Furthermore, there is the additional drawback of lack of *temporal isolation* among the activities, because a higher-priority task may indefinitely delay a lower-priority one. With reservation-based scheduling, each real-time activity needs to provide its own computational and timing requirements to the kernel, independently of what else is being run on the system. With *hard* reservations, whenever an activity tries to exceed its declared computational requirements, the OS/kernel stops it (*throttling*) till its next activation, even resulting in a non work-conserving scheduler (CPU can be left idle with real-time tasks under throttling). With *soft* reservations, the scheduler is work-conserving instead, and reservations can opportunistically gain extra allocation, either as left unused by others, or because the reservations do not saturate the system. In both cases, reservation-based scheduling ensures that the capability of each real-time task to meet its timing requirements depends only on the fact that the declared computational and timing requirements match with the actual ones at run-time, thus real-time applications are *isolated*. This constitutes a basis for stable task execution on top of which sound mathematical models can be built.

3.3. Deadline-Based Scheduling in Linux

Since version 3.14, the Linux kernel supports reservation-based scheduling through the `SCHED_DEADLINE` scheduling policy. `SCHED_DEADLINE` implements CPU

reservations using the Constant Bandwidth Server (CBS) algorithm [16], which is based on Earliest Deadline First (EDF) [49].

More precisely, `SCHED_DEADLINE` allows one to associate a task τ_i with three scheduling parameters: a `runtime` Q_i, a `deadline` D_i and a `period` P_i. This results in the Linux kernel granting the task Q_i time units on the processor every time window of duration P_i, where in each period the Q_i time units of execution are granted within the relative deadline D_i.

Focusing on the standard case of relative deadline equal to the period, the typical use of `SCHED_DEADLINE` is the one of a task with known minimum inter-arrival period T_i and worst-case per-activation execution time C_i, where one would set its CBS scheduling runtime as $Q_i = C_i$ and the scheduling period and deadline as $P_i = D_i = T_i$.

On multiprocessor systems, `SCHED_DEADLINE` implements a *global* EDF-based scheduling policy, but it can also be configured as a *partitioned* scheduler by proper use of the `cpuset`[7] CGroup controller. On SMP systems with CPU frequency locked, the global configuration of `SCHED_DEADLINE` guarantees each task in a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ to complete with a well-known worst-case tardiness beyond its relative deadline [50, 51], as long as the system capacity is not violated:

$$\sum_{i \in \Gamma} \frac{Q_i}{P_i} \equiv \sum_{i \in \Gamma} \frac{C_i}{T_i} \leq m, \tag{1}$$

with m being the number of CPUs. Note that, in the equation, $\frac{C_i}{T_i}$ represents the worst-case utilization of task τ_i, which, as due to the common setting for the reservation parameters described in the previous few paragraphs, is also equal to the reserved CPU bandwidth $\frac{Q_i}{P_i}$.

On the other hand, with the partitioned configuration, `SCHED_DEADLINE` guarantees each task to respect its relative deadline, as long as the capacity of each

[7] More information at: `https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt`.

Table 2: Parameters characterizing real-time periodic tasks and SCHED_DEADLINE reservations.

Symbol	Description
m	Number of CPUs in the system
n	Number of real-time tasks in the system
$\Gamma = \{\tau_1, \ldots, \tau_n\}$	Set of considered real-time tasks
Γ_j	Set of real-time tasks on CPU j
Q_i	SCHED_DEADLINE runtime for task τ_i
D_i	SCHED_DEADLINE relative deadline for task τ_i
P_i	SCHED_DEADLINE period for task τ_i
$\frac{Q_i}{P_i}$	Reserved CPU utilization for task τ_i
C_i	Worst-case per-activation execution-time of task τ_i
T_i	Minimum inter-arrival period of task τ_i
$\frac{C_i}{T_i}$	Worst-case CPU utilization of task τ_i

CPU $j \in \{1, \ldots, m\}$ is not violated:

$$\forall j \in \{1, \ldots, m\}, \sum_{i \in \Gamma_j} \frac{Q_i}{P_i} \equiv \sum_{i \in \Gamma_j} \frac{C_i}{T_i} \leq 1, \tag{2}$$

with $\Gamma_j \subseteq \Gamma$ denoting the tasks on CPU j.

The just introduced notation, summarized in Table 2, will be extensively used throughout the paper.

Furthermore, since version 4.16, SCHED_DEADLINE has been integrated with schedutil [39, 40] by using a variant of the GRUB-PA algorithm [37]. As a result, the scheduler is now able to scale the CPU frequency according to the bandwidth requirements of the SCHED_DEADLINE tasks, as well as the estimated utilization of tasks from the other scheduling classes, limiting the power consumption while preserving their timing constraints.

4. Background on Android

This section provides background information on: the Android audio pipeline architecture used to enable low-latency features for audio applications; how Android tries to achieve energy efficiency while scheduling latency-sensitive tasks; and deadline-based scheduling features in the Linux kernel. These concepts are at the foundation of our proposed solution.

4.1. Android Audio Architecture

Audio applications can be developed in Android by using, as shown in Figure 1, the high-level Java APIs available through the set of `android.media.*` classes, such as: `media.MediaPlayer`, to control playback of audio/video files and streams towards the local devices; `media.MediaRecorder`, to record audio/video from the local devices; `media.AudioManager`, to control volume and other playback tunables; `media.AudioTrack` to handle buffering of audio samples for playback; `MediaCodec`, to handle a plethora of available codecs for media playback and streaming. These Java classes interact via the Java Native Interface (JNI) with their C/C++ counterparts, which access the available audio services by interacting with the `AudioFlinger` server through Binder. Within `AudioFlinger`, up to 32 different audio streams coming from different applications are mixed by the `Mixer` thread, normally activating at a period greater than 20 ms, and having the ability to perform complex adaptations such as sample rate conversions. To avoid possible audio glitches, this thread runs with a boosted priority (using a negative `nice` value) within the `SCHED_OTHER` scheduling class. The `Mixer` thread hands over the audio samples to the underlying user-space Android audio Hardware Abstraction Layer (HAL), which ultimately sends the data to the device drivers through the `ALSA` sub-system within the Linux kernel.

Developers willing to realize low-latency, interactive audio applications, typically have to make the extra step of implementing a JNI component in their application, and use directly the available low-latency C/C++ audio APIs, either the traditional `OpenSL ES` (available since Android 4.1), or the recently added

AAudio (available since Android 8.1), or the further Oboe over-arching API that is capable of using either the former or the latter API. These APIs interact with a particular component of the AudioFlinger server, called FastMixer, that has a much shorter activation period, normally 2–5 *ms*, which in turn hands over audio data to the audio HAL and ultimately to the kernel. In order to ensure a glitch-free playback under such conditions, FastMixer keeps its functionality at the bare minimum (i.e., up to only 8 tracks can be mixed, one of which is the audio coming from the regular non-low latency path, and no resampling is supported), and it has a real-time thread scheduled using the SCHED_FIFO policy. In particularly demanding cases, it is possible to bypass the FastMixer entirely, so to avoid its overheads, by requesting exclusive access to the audio device, when initializing AAudio.

In what follows, the focus of this paper is on building low-latency audio applications making use of the AAudio framework, particularly for the specific case of exclusive access to the audio device[8].

4.2. Low-latency Audio Pipeline in Android

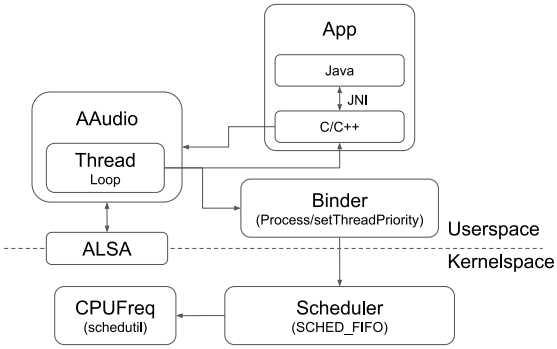


Figure 2: Logical blocks involved in a low-latency Android AAudio playback.

The low-latency audio pipeline in Android has the structure exemplified in Figure 2. A typical Java application that wants to use the low-latency audio

[8]A design suitable for the general case that includes an arbitrary processing graph of computations producing the audio stream is deferred to future work.

pipeline must use JNI to define the callback which generates the audio stream, and must export the callback to the audio stream framework through the provided API. When the audio stream starts, `AAudio` generates a new thread and sets its scheduling class to `SCHED_FIFO` through a Binder call. This thread loops forever executing an application-supplied callback, which produces the audio frames for playback through the `ALSA` subsystem. An audio frame contains a sample for each available channel (e.g., for stereo playback, a frame has two samples, for the left and right channels). Instead of using a blocking operation on the device (i.e., `select`, `poll` or `epoll`), the looping thread sleeps for an amount of time determined by a timing model that wakes up the process when there is sufficient room in the audio buffer to be refilled.

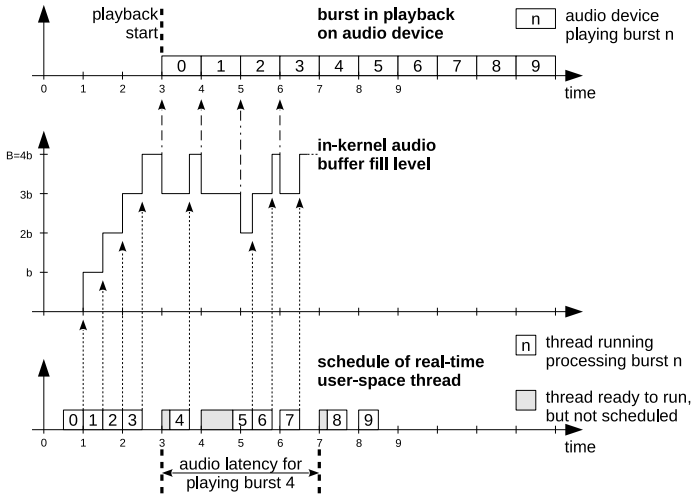


Figure 3: Exemplification of audio processing pipeline, showing the schedule of the real-time application thread processing each burst (on bottom), the fill level of the audio ring-buffer within the kernel (central) and the audio burst under playback at each time instant (on top). Time on the x axis is expressed as multiples of b/S.

The application callback writes audio frames in chunks called *bursts* of size b, that are queued into the playback ring-buffer of the audio device, which has a bigger size B set as a multiple of the *burst* size: $B = k \cdot b$, $k \in \mathbb{N}^+$.

To ensure a smooth and glitch-free playback, the playback ring-buffer is kept

Table 3: Typical audio parameters.

Symbol	Value (default)	Value (low-latency)	Description
S	48	48	Sampling rate (kHz)
b	480	64	Audio burst size (samples)[9]
B	7680	128–192	Audio buffer size (samples)[9]
r_t	160	2.67–4	Audio latency (ms)

as full as possible, as exemplified in the sample scenario depicted in Figure 3. The application has typically a ramp-up phase at the beginning of the playback (the first 4 bursts in the figure), during which the full B-sized buffer is filled up, followed by a nearly periodic activation, with additional b audio frames provided by the callback at each activation (as visible for bursts 4, 7, 8, 9 in the figure). With a configured sampling rate of S frames per second for the audio device, this results in one activation of the application callback every period of length b/S. However, B is the parameter which directly affects the audio latency, because the residency r_t of an audio sample in the device ring-buffer is: $r_t = B/S$ (as visually highlighted in the figure for burst 4).

When the real-time audio thread wakes-up, it may be scheduled immediately (i.e., at time $6b/S$ in the figure), or its execution may be postponed by the scheduler due to other activities on the system (i.e., at time $4b/S$ in the figure), like non-preemptible kernel sections, serving interrupts or scheduling higher priority tasks. As a consequence, the real-time thread may fail to compute its next burst within its next activation time. In this case, the in-kernel buffer fill-level goes lower than usual; however, this does not immediately result in an audio glitch thanks to the buffered additional bursts (highlighted in the figure for burst number 5). An audio glitch only occurs if the audio processing is delayed further, up to the hard deadline of B/b periods.

Typical values of these parameters are reported in Table 3 for non-interactive playback scenarios, as well as low-latency audio applications.

4.3. Power Management on Linux/Android

Since Android 8, the default `CPUFreq` frequency-scaling governor is `schedutil`, which adjusts the CPU frequency according to the utilization statistics computed by the Window-Assisted Load Tracking [52] (`WALT`) or, in some experimentations, the Per-Entity Load Tracking (`PELT`) algorithm [53]. This is done by tracking the utilization of active and sleep times of individual tasks (per-entity tracking), then using this information to estimate the active utilization, on each CPU, by summing up the utilization of tasks belonging to each scheduling class (`SCHED_NORMAL`, `SCHED_RT`, `SCHED_DEADLINE`). This way, `schedutil` is able to choose the minimum frequency for each CPU that seems sufficient to handle the overall load on the CPU, considering all of the tasks that are active within the various scheduling classes. `WALT` has been introduced as an alternative to `PELT` that showed a better behavior on asymmetric Android devices, as explained in [52], where the two heuristics are compared in detail.

Differently from the Android fork, in the mainline Linux `schedutil` governor, these utilization metrics are either not present (`WALT`) or not used (`PELT`) instead, when it comes to choose the frequency to run a `SCHED_RT` or `SCHED_DEADLINE` task. For these tasks, `schedutil` always sets the maximum available frequency. The mainline behavior is particularly inefficient for system-on-a-chip (SoC) architectures that organize processors in clusters, where cores of the same cluster share the same frequency: even a single `SCHED_RT` or `SCHED_DEADLINE` task on a single core forces the whole cluster at the maximum frequency.

As different devices have different capabilities in CPU frequency scaling and power management (e.g., big.LITTLE[10], DynamIQ[11]), the Energy Aware Scheduling (EAS) [54] framework has been recently added to the Linux kernel to

[9] Device-specific values.

[10] More information at: `https://developer.arm.com/technologies/big-little`.

[11] More information at: `https://developer.arm.com/technologies/dynamiq`.

provide a unified view of these capabilities[12]. EAS manages metadata about the frequency clusters topology, along with information about power consumption and processing performance (called *capacity*) for each Operating Performance Point (OPP) of each CPU. EAS also includes mechanisms for exploiting this information at its best regarding task placement and OPP selection, based on the current workload.

Since the CPU frequency can only be chosen among a set of predefined OPPs imposed by the hardware and, sometimes, further reduced by software constraints[13], the process of translating the utilization to the CPU frequency is performed by selecting the smallest OPP capable of satisfying the computational demand.

5. Proactive Adaptation of Reservations on Android

In this section we detail our proposal for engineering low-latency audio applications on the Android platform. We advocate the use of the `SCHED_DEADLINE` real-time scheduler, with `GRUB-PA` extensions, along with the `schedutil` governor. Second, we deal with dynamic audio workloads by proposing an extension to the `AAudio` API with a method that allows applications to declare their time-varying workload demand. The approach is implemented and validated with the use of `SynthMark`[14], a recently available open-source audio pipeline emulator for Android.

5.1. Deadline-Based Scheduling for Low-Latency Audio

This work focuses on low-latency audio applications making use of a single, sequential, non-suspending callback, and the real-time thread spawned by `AAudio` that executes the callback, then blocks until there is room in the audio buffer. The application may have other non real-time threads for its user interface

[12] More information at: `https://developer.arm.com/open-source/energy-aware-scheduling`.

[13] The vendors often provide access to a subset of the hardware OPPs depending, for example, on thermal dissipation capabilities, or energy availability of the final product.

[14] https://github.com/google/synthmark/.

(UI), but since they are not part of the playback latency sensitive path, they are ignored in this work. Interactions among the real-time and non-real time threads, if any (e.g., for passing configuration parameters from a UI), are usually carried out through lock-free protocols. This application structure represents a typical model for the majority of low-latency, interactive audio applications that are nowadays developed.

As explained in Section 4, the callback is activated every time the audio buffer has enough space to allocate at least a burst. This, coupled with the attempt to keep the buffer as full as possible, results in the activation of the real-time audio thread at every period of duration $T = b/S$. Each such activation needs to complete ideally before the next activation period: when a burst is put in playback onto the device, another burst is requested to be produced. Therefore, each activation of the real-time thread has a relative soft deadline of $D = b/S$, where the hard deadline is B/S instead.

We can thus apply the implicit deadline task model, as extensively studied in the real-time literature, and specifically, the CBS algorithm setting the scheduling period and deadline equal to $P = D = b/S$, and setting the scheduling runtime equal to the task worst-case execution time $Q = C$. The execution time C, needed for the processing of each callback activation, is a value much harder to characterize. It depends on: 1) the computational capabilities of the hardware architecture where the application is running; 2) the maximum capacity of the CPU the real-time thread is scheduled on, whenever CPUs can have asymmetric performance like in the case of Arm `big.LITTLE` architectures widely used in mobile Android devices; and 3) the CPU frequency on architectures supporting DVFS, as dynamically adapted by the scheduler and `CPUFreq` subsystem.

The intrinsic dynamics of the application workload has also to be considered. Even on a CPU running at a constant frequency, an application can have very different computational demands. This may depend, for instance, on the number of musical notes played by the user for a virtual instrument, or the number and complexity of active audio filters/effects for an audio processing application.

Taking as a reference a virtual instrument synthesizer, such as a virtual piano, because of the use of a sustain pedal or because of the notes decay, the number of notes that are simultaneously played can dynamically vary in a typical range between 0 to a few hundreds, resulting in a callback workload that undergoes quick variations spanning across multiple orders of magnitude.

The computational requirement C is a fundamental parameter, that can be used to: 1) analyze in advance and guarantee the schedulability of the tasks set according to the real-time theory, 2) allocate the proper amount of resources to the process, and 3) determine the most efficient CPU frequency and, for heterogeneous architectures, the most efficient processor the task shall run on. The next section outlines how we tackled all these challenges.

5.2. Adaptive Scheduling

To schedule the audio callback, Android currently uses `SCHED_FIFO` along with either `PELT` or `WALT` as signals to drive the OPP selection via `schedutil`.

One of the main limitations to using this approach for low-latency audio applications, is related to the reactive OPP selection policy these signals enforce. Indeed, when either `PELT` or `WALT` are in use, when the audio workload suddenly increases, it takes some time for the signal to detect the new CPU bandwidth requirement correctly and trigger a frequency change. In the worst case, it could take between 50 ms and 100 ms for `PELT` (depending on its configuration) to detect a 90% increase of the CPU bandwidth demand. For a low-latency audio scenario where $B = 256$ and $b = 64$ at $S = 48\ kHz$, the `PELT` detection latency is too high when compared to the buffered audio of only 5.33 ms. Even when `WALT` is in use, a 90% CPU utilization demand increase cannot be detected in less than 10 or 20 ms, depending on its configuration. Thus, independently from the use of `PELT` or `WALT`, with just kernel space driven OPP selection, it is hard to grant a glitch-free playback meeting the requirements of a dynamic-workload, low-latency audio application.

The reactive nature of CPU frequency driving from kernel space cannot be easily fixed without explicit hints from user-space. This is why a possible so-

lution has been so far to control the frequencies from user-space by explicitly setting constraints on the minimum frequency the kernel can choose. Such approaches unfortunately manage to meet the tight requirements of low-latency applications, sacrificing the capability of the system to keep a low power consumption profile. Also, they lack a specific API to define bandwidth requirements in a platform-independent way. Platform independence is of paramount importance when it comes down to building portable applications which can still satisfy tight temporal constraints independently from the specific platform on which they will be executed.

To tackle all these problems, we propose in this paper to extend the `AAudio` API to allow the application to notify the underlying OS about imminent workload changes. This is done by dynamically supplying a `workUnits` parameter, whose variations correlate with the expected workload changes of the real-time thread callback. The `schedutil` governor can then immediately trigger an up-frequency switch, even before starting the heavier computations, resulting in the needed CPU frequency increase being anticipated, so to efficiently and adequately support the increase in processing power demand. This mechanism allows audio applications to inform the OS about expected workload changes so that the system can adjust the OPP to a value that satisfies the QoS requirements of the application, preserving the critical objective of minimizing the energy consumption. Note that this mechanism works in conjunction with the existing OPP selection policy for the other scheduling classes, as `schedutil` sums up the active utilization of `SCHED_DEADLINE` tasks, that is influenced by the described `workUnits` notification mechanism, with the estimates made independently for tasks belonging to the other scheduling classes, and then picks the minimum OPP able to satisfy the overall demand.

A simplified representation of this architecture is presented in Figure 4, showing a soft-synthesizer audio application that receives as an input (e.g., from the touchscreen or via a MIDI peripheral) a new number of requested notes to play or a different filter configuration for audio processing. This input information is used to update the internal application configuration and to notify

the system of the future workload as `workUnits`, a number that for example can represent the number of played notes if playing samples or an estimation of the computational effort introduced by each active filter in an audio synthesizer. In order to ease development of an audio application across a multitude of different devices, as commonly needed for Android applications, the `workUnits` hint is provided in a platform-independent way, and the framework maps its values to resource demands as monitored on-line during the application execution.

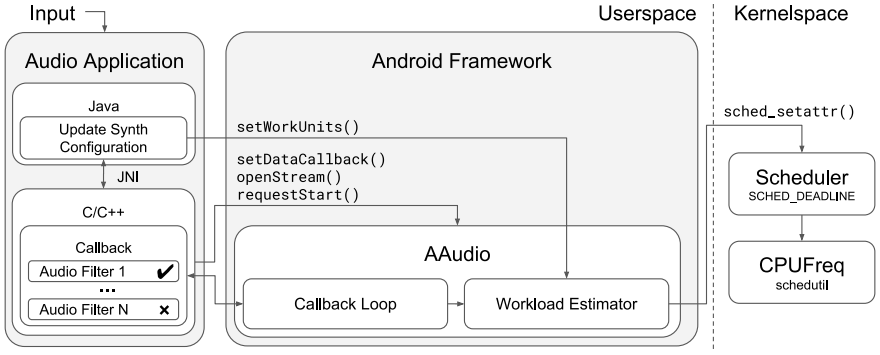


Figure 4: Example block diagram of an audio application using the dynamic bandwidth allocation API.

The translation from `workUnits` to the actual OPP can be designed by preserving the `SCHED_FIFO` scheduling policy, but forcing the `CPUFreq` governor to use a user-space defined utilization for the given task, that would directly be translated to a minimum CPU frequency value. The Linux kernel does not support this `schedutil` extension yet, but it is currently under discussion on LKML[15].

In this paper, we undertake an alternative approach, based on the use of the `SCHED_DEADLINE` scheduling class, as shown in Figure 5. The `AudioTask` notifies the currently requested `workUnits` to the `HostCPUManager` which, extending concepts from [9, 34], internally uses a `Predictor`[16] to estimate the upcoming computing

[15] https://lwn.net/Articles/751361/.

[16] In [9, 34], we experimented with feedback-based control strategies with a split of functionality between a *predictor* component, responsible for estimating the upcoming task instance execution time requirements, and a *controller* component, that uses the latter information to

time of the callback according to statistics of the past observed processing times, and by modulating the obtained value through the supplied `workUnits` values.

To produce valid heuristics, the `Predictor` requires reliable and precise measurements on the callback runtime. Measuring this time by executing the `clock_gettime()` syscall before and after running the callback, and finally subtracting the two results is not a reliable approach. In fact, the final value can be affected by both the CPU frequency changes happened during the callback execution, and the capacity of the CPUs where the task executing the callback has been executed, e.g., a `big` or `LITTLE` CPU. Both these problems have been solved by extending the `sched_getattr()` syscall to return the next absolute deadline d and the remaining CBS server runtime q, where the scheduler already scales this last value according to the CPU frequency and capacity. These values can be sampled before and after the callback execution, thus, by also knowing the actual CBS computing time Q that the task can use every period T, we can compute the *normalized* execution time C_m of the callback as:

$$C_m = q_{\text{start}} - q_{\text{end}} + Q \left\lfloor \frac{d_{\text{end}} - d_{\text{start}}}{T} \right\rfloor . \tag{3}$$

Here, the rightmost term is almost always zero: whenever the callback completes within its relative deadline $D = b/S$, according to the CBS rules the absolute deadline is not updated, so $d_{\text{end}} = d_{\text{start}}$. For those rare cases in which the callback completes beyond its (soft) deadline (e.g., burst 5 in Figure 3), at the end of the callback, the CBS algorithm will have recharged the budget of a quantity equal to Q for each deadline postponement of the scheduling period $P = T$.

After each task activation callback completes, its normalized runtime C_m is measured, and the `Predictor` updates its stored runtime statistics. The estimated runtime for the next activation $k + 1$, $C_e[k + 1]$, is computed based on

compute the reservation runtime needed to meet various desirable control goals. This prior experience in control strategies and software architectures for adaptive scheduling of multimedia applications, constitutes useful background that helped us shaping our proposed new proactive adaptation strategy for this paper.

the past observed samples. This can be done in a variety of ways. For example, in our prior works [12, 11, 9], we have been using and comparing various techniques for workload estimation in multimedia use-cases, including moving averages, percentile estimators and others, possessing different computational and memory demands, and resulting in different properties and behaviors for the closed-loop system. In this work, we chose to use a particularly simple yet effective predictor, implementing an exponentially weighted moving average

$$C_e[k+1] = \alpha C_m + (1-\alpha) C_e[k], \tag{4}$$

with *asymmetric smoothing constants* α: if the C_m value is bigger than $C_e[k]$, then $\alpha = 0.95$ is used, otherwise $\alpha = 0.1$ is used. This allows for a fast reaction on sudden workload growth (OPP increase), and a slower reaction in response to workload decreases (OPP decrease). The predictor stores an independent moving average for every observed `workUnits` value within a hashmap, where at each callback completion the stored runtime value associated to the current `workUnits` is updated according to Equation (4). The advantages of using a hashmap are that it does not require prior knowledge of the minimum or maximum value of `workUnits` (as an array would need), and guarantees efficient access and modify operations. When the `Predictor` is asked to estimate the next activation computing time, the following cases may happen:

- the hashmap is empty: the maximum computing time, corresponding to a configurable constant bandwidth, is returned. For example, 0.94 has been used in our experiments.
- `workUnits` found in the hashmap: the value stored in the hashmap is returned.
- `workUnits` smaller than any other value in the hashmap: to be conservative, the duration for the smallest `workUnits` value in the hashmap is used.
- `workUnits` bigger than any other value in the hashmap: the returned computing time is computed by intersecting the linear regression among the

available elements in the hashmap. In absence of any other information on the processing workload, we make the assumption that the callback computing time is likely a linear function in the number of `workUnits`, for simplicity.

The pessimism of this algorithm prevents audio glitches, as higher OPP than strictly needed are used at application start-time, when the workload statistics are not available to the `Predictor` yet, and tasks are assigned to CPUs respecting Eq. (2). Note that alternatives are possible for the just presented heuristic. Indeed, in our prior investigations [13], we proposed a "FQDB Library", that used to have a SQLite-based persistent storage of the resource requirements for various applications, that could either be populated on-line during normal operation, or – more interestingly – during a separate profiling phase. This would consist in running real-time code excerpts of the application right after installation on a device. In the same works, both linear and custom, application-supplied, non-linear interpolation was proposed, in order to support the framework in figuring out the right resource allocation for those points in the configuration space that occur anyway for the first time, and for which we would not have information in the persistent storage. However, further discussion of these details are left out of the present work, due to space reasons.

The `BandwidthAllocator` receives the latest estimated C_e value, which is used to update the `SCHED_DEADLINE` runtime Q. Since the predicted C_e derives from a slightly modified average value of the previously measured runtimes, the noise in the measurements may still cause underestimations in the bandwidth computation, so some margins are added to the value that is finally used to update the parameters of the scheduler:

$$Q = m_m C_e + m_o, \tag{5}$$

where $m_m \geq 1$ is a configurable proportional margin and $m_o \geq 0$ is an offset. This operation is performed every time the application updates the `workUnits` number, and after a given number of written bursts, e.g., 30 in our experiments. Thanks to the use of the GRUB-PA policy, the new `SCHED_DEADLINE` workload is

immediately communicated by the scheduler to `schedutil`, which takes care of updating the CPU frequency accordingly.

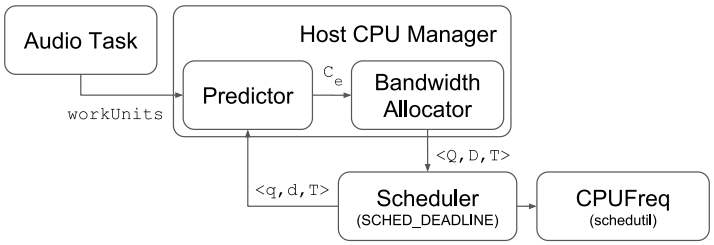


Figure 5: Dynamic bandwidth allocation: logical blocks.

Summarizing, our proposed framework enables Android low-latency audio applications relying on the `AAudio` API to automatically leverage predictable execution and temporal isolation provided by the `SCHED_DEADLINE` scheduler in the Linux kernel. With no code changes, the application is attached a CPU reservation with parameters that are automatically set by the framework. Additionally, for applications with workload expected to undergo significant and abrupt changes (e.g., a synthesizer with a variable number of voices), developers can leverage a new API notifying workload changes as a `workUnits` signal, which has been designed to be a platform-independent means for notifying the kernel of the expected relative variability of the workload in upcoming processing cycles.

5.3. SynthMark

`SynthMark` is a benchmarking tool that implements a real audio synthesizer generating an audio stream that, instead of being sent to the real Android audio pipeline, is sent to a virtual audio pipeline and consumed by a virtual audio sink, both embedded in the tool. The internals of `SynthMark` have been designed for realistic emulation of the behavior of an Android audio application using the real low-latency audio pipeline features. `SynthMark` has been designed and developed to provide more flexibility for Android OS developers when customizing the audio subsystem concerning configurations, programming models,

and scheduling algorithms, and allows for obtaining detailed and comprehensive performance statistics so that it becomes easier to identify in advance strengths and weaknesses of the approaches under development. Being `SynthMark` platform independent, it also allows for measuring the effects of the underlying kernel on latency, independently of the Android audio framework.

As shown in Figure 6, what in Android would be the audio application, in `SynthMark` is implemented by the `Synth` module that is, in fact, a real polyphonic audio synthesizer that produces audio samples generated with a chain of oscillators, filters and ADSR (Attack, Decay, Sustain, Release) modules, and whose computational effort depends on the number of requested notes. The `Synth` module exports a callback that is quasi-periodically executed by the `VirtualAudioSink` module, which implements a thread whose scheduling parameters can be modified and, by default, on Linux kernel based systems, uses `SCHED_FIFO`. After the application callback is executed, `VirtualAudioSink` performs a write operation on the (virtual) audio sink to notify the availability of new audio samples, resulting in a blocking operation that waits until the buffer has enough space to contain the newly produced data. `VirtualAudioSink` also counts under-runs occurring whenever the audio buffer gets empty.

After every callback execution, the `VirtualAudioSink` measures its duration and forwards this value, together with the current number of notes played, to the underlying `HostTools` component, to keep track of workload changes. `HostTools` takes care of storing callback statistics and adjusting the bandwidth through internal heuristics. It will be described and evaluated in the next section.

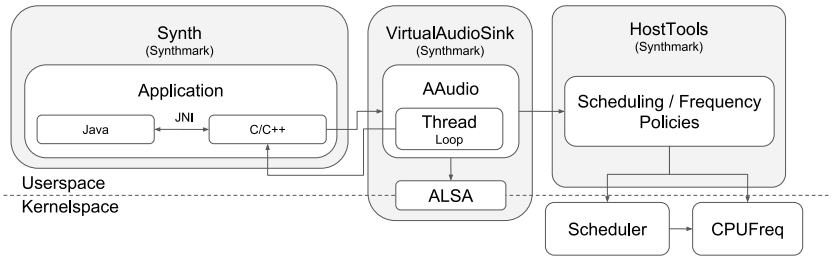


Figure 6: `SynthMark` logical blocks in relationship with the Android low-latency audio pipeline.

6. Experimental Results

To validate the proposed solution and evaluate its performance compared with the current Android approach on both energy efficiency and audio latency, we performed a set of experiments using an extension of the `SynthMark` tool, which is the de-facto standard used by Android developers to benchmark the audio subsystem. The extended `SynthMark` tool is available at `https://github.com/balsini/synthmark/tree/JSS-2018`. The experiments are run on a HiKey 960 board[17], which embeds a big.LITTLE SoC with DVFS features. This board, along with others from HiKey, constitute ideal development and prototyping boards for Android-based development, because they possess the typical hardware set-ups that can be found in nowadays and future upcoming mobile Android-based platforms, with advanced power management features as coming from the adoption of big.LITTLE architectures. Also, they come with a complete set of drivers for the Android branch of the Linux kernel. The reference board was set up with our extended Android Linux kernels[18]. While the audio latency performance was internally measured by `SynthMark`, the energy consumption was physically measured with an ACME CAPE[19] energy meter connected to the HiKey 960 board. This device allows for a programmatic access to power consumption measurements throughout an experiment, so that the needed readings can be automatically gathered from the software tools driving the experimentation. On the software side, the `SynthMark` tool has been used by pinning it to a `big` CPU. This was done to ensure fair comparisons by ruling out possible side-effects due to tasks migrations and CPU capacities variations among different cores. However, this set-up closely mimics the single-threaded nature of several low-latency audio processing applications, which normally have a single real-time thread taking care of audio processing, and other threads inter-

[17] More information at: `https://www.96boards.org/product/hikey960/`.

[18] `https://github.com/balsini/linux/tree/JSS-2018-android-hikey-linaro-4.9`, and `https://github.com/balsini/linux/tree/JSS-2018-android-hikey-linaro-4.9-dl-integration`.

[19] More information at: `https://baylibre.com/acme/`.

acting with the user for monitoring, control and configuration purposes, which are not time critical and can be run at non real-time priority.

In the following, we report experimental results carried out over the above mentioned set-ups, describing first a few calibration and tuning experiments that were needed as preliminary operations, then highlighting the advantages of the proactive adaptation mechanism in terms of OPP adaptation latency, in the particularly challenging scenario of a step in the expected workload, and finally in a generic scenario involving the execution of SynthMark with a randomly varying CPU workload.

6.1. Calibration and Tuning

In a preliminary calibration phase, the `VoiceMark` benchmark of `SynthMark` has been used to evaluate the total CPU utilization for a given number of active audio voices. We found out that 210 voices were able to saturate our `big` CPU when running at the maximum frequency.

The following experiments used the `LatencyMark` benchmark of `SynthMark`, with a burst size of $b = 64$ and a number of `workUnits` that dynamically varied every 3 seconds.

An initial tuning phase was aimed at evaluating the total audio latency required when using the current Android low-latency approach based on `SCHED_FIFO` and `WALT`, with its default window size of 20 ms. Note that this experiment is used only for tuning the real-time performance of `SynthMark`, so no power consumption has been measured throughout its execution. This evaluation consisted of alternating the number of voices from 5 to 185, corresponding to utilization of the `big` CPU ranging from 2.38% to 88%, representing the most challenging workload pattern for the heuristics used by the kernel scheduler. As shown in Figure 7a, the `WALT` utilization followed the real demand as due to the increase in the number of voices, reaching the final steady state with a delay of approximately 80 ms because of the `WALT` window size, which also postponed the frequency update. In this example, the system was able to reach the proper OPP after about 60 ms after the workload increase, about 20 ms before the

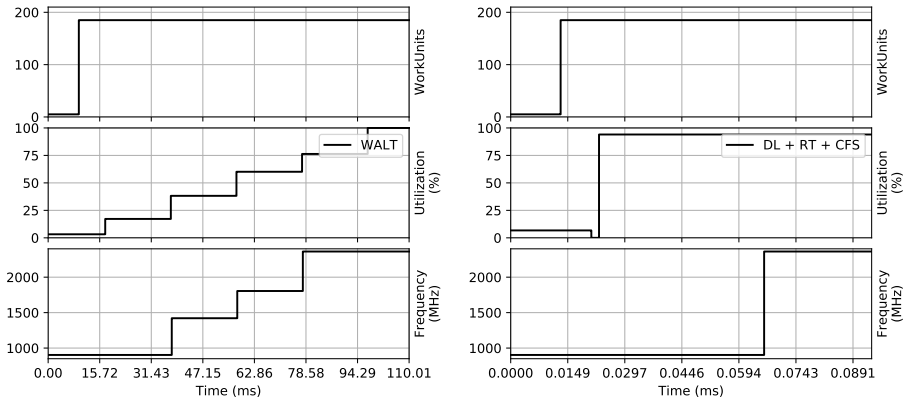
steady state condition, since the CPU utilization sampled at that time already corresponded to the final OPP. Because of this frequency adaptation delay, when the application load increases, the system is not capable of instantly satisfying its computational requirements, thus generates audio glitches with relatively small buffer sizes.

With the previously described configuration, a buffer size of 20 bursts was necessary to achieve smooth audio playback, that on the other hand introduced an audio latency of approximately 26.67 ms. When the workload decreases instead, the system kept an OPP higher than required for a small amount of time, resulting in a modest waste of energy.

6.2. OPP Adaptation Latency

The next experiment was run with `WALT` disabled and shows how, by dynamically adapting the task bandwidth according to the number of `workUnits` and the callback duration, `SCHED_DEADLINE` can automatically adjust the OPP. This experiment has the same configuration previously shown, with a periodic `workUnits` variation between 5 and 185. The safe margins of the `BandwidthAllocator` are used to avoid underestimations of the required bandwidth to compensate the measurement noise and the uncertainty of the upcoming execution times, and have been empirically tuned after preliminary testing as $m_m = 1.005$ and $m_o = 0.041$. Figure 7b presents the experimental results for this scenario. In this case, since `WALT` was not active, `schedutil` evaluated the required CPU bandwidth as a sum of the `PELT` utilization reported by the `SCHED_NORMAL` (CFS), `SCHED_FIFO` and `SCHED_RR` (RT), and `SCHED_DEADLINE` (DL) scheduling classes. In our experiments, `SynthMark` was the only application running on the system; thus, the sampled `PELT` utilization had almost no interference from other tasks, except interrupt handlers or background activities. This plot shows that the new deadline-based approach with dynamic bandwidth reservation was able to raise the system utilization in less than 0.01 ms and the frequency adjustment is requested in approximately 0.054 ms after the workload change. By adding the additional time required by the CPU to switch the OPP on our HiKey

960 platform, this approach required about 0.45 ms to complete the frequency adaptation after the workload increased.



(a) (b)

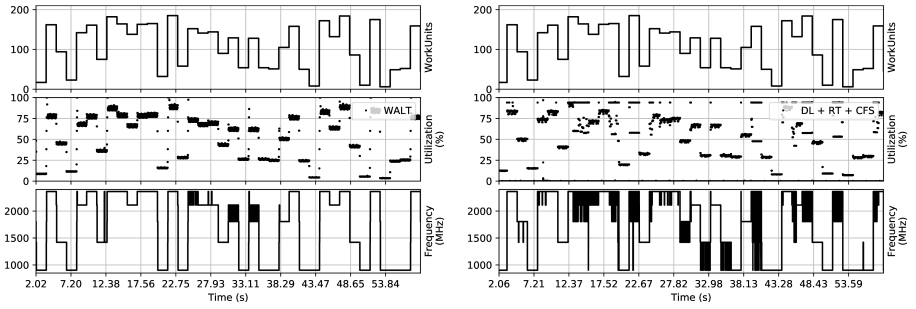
Figure 7: `SynthMark` utilization perceived by `schedutil` using `SCHED_FIFO` with `WALT` (7a) and by the `Predictor` (7b), and frequency adjustment at varying `workUnits` values.

Since the OPP update is almost immediately performed before the workload change, the callback runs at a CPU frequency that satisfies its timing requirements. The profitability of this behavior for the reactiveness of the audio task was demonstrated by `LatencyMark`, which returned that the system is stable with a buffer size of 2 bursts, corresponding to 2.67 ms.

6.3. Randomly Varying Workload

In order to evaluate the quality of this approach in more generic and realistic workload scenarios, the following experiments were run to evaluate `SCHED_FIFO` with `WALT` and the adaptive bandwidth allocation with `SCHED_DEADLINE` approaches by running `LatencyMark` with a random number of `workUnits` at every step, still in the range between 5 and 185, for a total duration of 60 seconds. The seed of the random number generator has been fixed to allow the reproducibility of the experiment and to provide the same `workUnits` random sequence when comparing the two approaches.

Figures 8a and 8b show the tight relationship between the CPU frequency and the number of `workUnits` in both the `SCHED_FIFO` with `WALT` and the adaptive bandwidth allocation with `SCHED_DEADLINE`. If the adaptive bandwidth allocation approach can keep the audio latency at 2.67 ms, on the other hand, the estimation of the utilization is more pessimistic than the one performed by `WALT`.



(a) (b)

Figure 8: `SynthMark` behavior using `SCHED_FIFO` with `WALT` (8a) and with adaptive bandwidth allocation (8b), with randomly varying `workUnits` values.

For the sake of completeness, we also tested the current solution implemented by many low-latency audio application developers, that force the CPU frequency to stay fixed at the maximum allowed value, and using `SCHED_FIFO`, similarly to what happens in a mainline Linux kernel when any `SCHED_RT` or `SCHED_DEADLINE` task is active. Concerning the audio latency, this solution achieved the same results of the adaptive bandwidth allocation approach, reducing the delay to 2.67 ms, but forced the CPU to run at its maximum OPP, also when not required.

Our proposed proactive adaptation of the computational bandwidth with `SCHED_DEADLINE` reflects on the CPU frequency selection that, as shown in Figure 9, tends to prefer higher CPU frequencies compared to `SCHED_FIFO` using `WALT` with dynamic `workUnits`, while the fixed workload approach always uses the maximum OPP. Being the single core power line not accessible, a direct measurement of how this behavior affects the energy consumption of a single core cannot be performed on the device. Instead, it is possible to derive a rough

estimation by using the official, normalized power consumption metrics included in the Device Tree of the kernel[20], that are used by the energy model implemented in the Linux kernel to find the most efficient task placement among the available CPUs of the device. For each analyzed frequency governor approach W, given the normalized power consumption in the Device Tree as P_f and the measured residency time of the task when using the governor W within the frequency f as t_f^W, the normalized energy consumption $E_N(W)$ has been estimated as:

$$E_N(W) = \sum_f t_f^W P_f.$$

As summarized in Table 4 and visually depicted in Figure 9, it turned out that, when using `SCHED_FIFO` with the standard `WALT` based frequency adaptation, the energy consumption of the CPU was approximately 96.16, but under these conditions the achieved audio latency is poor ($26.67ms$). Low-latency audio at $2.67ms$ can be recovered in standard Android by keeping the frequency fixed at its maximum, realizing a much greater energy consumption of the CPU of nearly 172.71. Using the adaptive bandwidth approach with `SCHED_DEADLINE` as proposed in this paper, instead, we can achieve both a glitch-free low-latency playback at $2.67ms$, and realize a significant energy saving, with a CPU energy consumption of 102.17.

6.4. Measured Energy Consumption

Finally, we provide a real measurement of the total energy consumption of the HiKey 960 board, obtained with the ACME CAPE energy meter directly connected to the board power supply, to compare the efficiency of the different approaches. Clearly, we expected the meter to report higher energy consumptions for the overall platform, in corrispondence of higher energy consumption

[20]The energy consumption metrics for HiHey boards can be found at the following link: `https://android.googlesource.com/kernel/hikey-linaro/+/android-hikey-linaro-4.9/arch/arm64/boot/dts/hisilicon/hi3660-sched-energy.dtsi`. Please, refer to `CPU_COST_A72` field for the energy consumption of an HiKey 960 active `big` CPU.

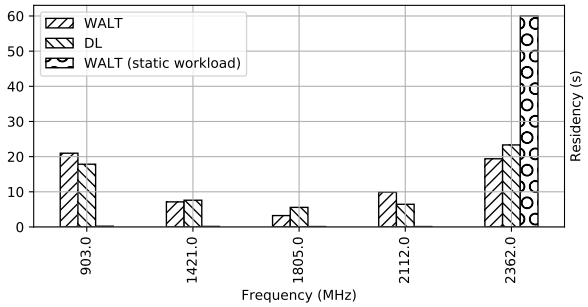


Figure 9: Frequency residency comparison between the `WALT` and the adaptive bandwidth allocation approaches. The detailed data is shown in Table 4.

for the CPU only, estimated as described just above. Indeed, it turned out that when using `SCHED_FIFO` with `WALT`, the total board energy consumption was 206.35 Ws (Watt $\cdot$ second) while, when forcing the CPU frequency to stay fixed at the maximum OPP, the total energy consumption reached about 256.93 Ws. With our proactive adaptation approach using `SCHED_DEADLINE`, the board had a total energy consumption of 210.22 Ws instead.

As summarized in Table 5, by using `SCHED_FIFO` with `WALT` it is possible to either reduce the energy consumption increasing the latency (`WALT` with dynamic `workUnits`) or reduce the latency increasing the energy consumption (forcing the CPU at the maximum OPP). It is not possible to achieve low latency with low energy consumption. On the other hand, using our proposed adaptive bandwidth allocation with `SCHED_DEADLINE`, it is possible to achieve a low latency (2.67ms, the same latency obtained with CPU blocked at its maximum OPP) while consuming a small amount of energy (210.22 Ws, only slightly larger than the energy consumed by `WALT` with dynamic `workUnits`). In other words, *our proposed approach outperformed what was achieved under* `SCHED_FIFO` *using* `WALT`, with a 40% reduction in the CPU energy consumption for the same latency.

Table 4: Frequency residency comparison and estimation of normalized CPU energy consumption based on the kernel energy model data, for the WALT and the adaptive bandwidth allocation approaches.

Frequency	Normalized power consumption from Device Tree	Frequency residency (s) SCHED_FIFO with WALT (dynamic)	SCHED_FIFO with CPU at max frequency	Adaptive bandwidth
903	404	20.976	0	17.845
1421	861	7.133	0	7.608
1805	1398	3.240	0	5.568
2112	2200	9.903	0	6.452
2362	2848	19.391	60.644	23.325
Normalized energy consumption		96.158	172.713	102.168

Table 5: Summary of the audio latency and energy consumption obtained under the SCHED_FIFO with WALT (unmodified Android), with the CPU statically blocked at its maximum frequency, and the adaptive bandwidth allocation approaches.

	SCHED_FIFO with WALT (dynamic)	SCHED_FIFO with CPU at max freq.	Adaptive bandw.	Improv. wrt WALT (dyn.) (%)	Improv. wrt max freq. (%)
Audio Latency (ms)	**26.67**	2.67	**2.67**	**90%**	0%
CPU energy Cons.	96.16	**172.71**	**102.17**	−6.25%	**40%**
Total energy Cons. (Ws)	206.35	256.93	210.22	−1.88%	18.18%

7. Conclusions

This paper presents an approach to reduce the audio latency for professional grade multimedia applications in the specific context of the Android OS and its support for the execution of this type of applications. This is achieved by extending the Android API with 1) a mechanism to provide proactively hints (`workUnit`) on the expected workload change of the audio application, 2) a subsystem that forecasts the application computing requirements through heuristics combining prior observations with the supplied `workUnit` hint, and 3) an extension to `SCHED_DEADLINE` to return execution time measurements for the latency sensitive task, scaled according to both the frequency and capacity of the CPU on which the task was executed.

In the performed experimentation, carried out with an audio latency benchmark running on a single CPU, the presented solution outperforms the traditional energy efficient approach by reducing the audio latency by ten times, at the cost of a limited energy consumption increase of approximately 6.25%. Moreover, it provides a considerable energy consumption reduction of almost 40%, achieving the same audio latency with respect to the traditional low-latency approach.

8. Future Work

As regards possible future work, a more extended evaluation is planned, with more use cases and concurrent multi-threaded workloads, as well as tweaking further the internal parameters that are available in our proposed framework and the proposed runtime adaptation logic. For example, there might be constructive and effective means for mixing the proactive adaptation approach proposed herein, with other feedback-based mechanisms for the estimation and prediction of the runtime needed under various workload conditions [8, 9, 12]. This way, we plan to gain a more comprehensive assessment of the advantages of the proposed technique with a wider set of workloads, and possibly further enhance its effectiveness. Another possibility is the design of more effective means for

providing a starting reservation runtime to the scheduler, for example merging with what we proposed in some of our prior investigations [13].

We also plan to extend the approach to other Android sub-systems, for example addressing use-cases including a multitude of real-time tasks, related to, e.g., the audio processing pipeline, the video processing pipeline, along with other communication-related real-time tasks. This is expected to be manageable, as the envisioned approach leverages `SCHED_DEADLINE` that already supports scheduling and isolation of multiple tasks over multi-processors. We expect for example to gain an advantage from the application of the hierarchical extension to `SCHED_DEADLINE` we recently posted on LKML[21]. Additional issues that would arise in this context are related to designing Android-specific meaningful ways to deal with possible temporary overload situations. For example, we might need to integrate, adapt and expand the *supervisor* approach as envisioned in [12].

Additional directions of future research regard also extending the proposed technique so as to effectively deal with additional hardware platforms with novel power management features that significantly deviate from the model of the HiKey 960 device used in this paper. For example, the recently announced Arm DynamIQ architecture[11] promises to bring significant power-saving advantages over the already established and widely deployed `big.LITTLE` CPUs, but the proper consideration of such platforms is expected to need extensions to our proposed scheme. Moreover, other recently available platforms exist that manage to achieve excellent energy savings when idling the CPU, so letting tasks complete as fast as possible at the maximum frequency and maximizing CPU idle time slices is claimed to realize competitive energy consumption, as compared to DVFS-based approaches. An important comparison that has to be considered is then between our frequency scaling solution and such a *race-to-idle* scheduling policy.

Another critical aspect that will be considered is that, when in Android there are multiple coexisting low-latency audio applications, their samples are sent to

[21] More information at: `https://lkml.org/lkml/2017/3/31/658`.

the `FastMixer`, which performs the mixing and outputs the obtained signal to the audio device. Multiple interdependent tasks that need to complete within a precise deadline are also present in audio applications that exploit the available multi-processor features, using a parallel programming paradigm for audio processing. These scenarios may be effectively tackled through further extensions of `SCHED_DEADLINE`, as well as more complex techniques for computing the needed scheduling parameters for not having deadline misses, based on more complex analyses of concurrent real-time direct-acyclic graphs (DAGs) of computations. These are planned to be investigated in future works.

Finally, a further extension idea might be the one to consider, as done in our recent preliminary investigation [55], that the power consumption of a modern complex CPU while processing depends in a non-negligible way on the workload type that is being run. So, this may add one further dimension and further complexity to the already challenging problem of scheduling real-time tasks (along with non-real time ones) on multi-core, DVFS-capable heterogeneous mobile platforms, in an energy-efficient way. Whether or not an efficient mechanism dealing with this additional detail level is worth to pursue, would need further investigation.

Acknowledgements

We would like to thank the anonymous reviewers for having provided detailed and precise comments, allowing us to improve the clarity and readability of this paper.

References

[1] G. Sigismondi, Chapter 37 - personal monitor systems, in: G. M. Ballou (Ed.), Handbook for Sound Engineers (Fourth Edition), fourth edition Edition, Focal Press, Oxford, 2008, pp. 1413 – 1435. doi:https://doi.org/10.1016/B978-0-240-80969-4.50041-9.

URL http://www.sciencedirect.com/science/article/pii/B9780240809694500419

[2] D. M. Howard, J. Angus, Acoustics and Psychoacoustics, 2nd Edition, Butterworth-Heinemann, Newton, MA, USA, 2000.

[3] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the Linux kernel, Software: Practice and Experience 46 (6) (2016) 821–839, spe.2335. doi:10.1002/spe.2335.

[4] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa, Resource kernels: a resource-centric approach to real-time and multimedia systems, Vol. 3310, 1997, pp. 3310 – 3310 – 15. doi:10.1117/12.298417.
URL https://doi.org/10.1117/12.298417

[5] J. A. Stankovic, C. Lu, S. H. Son, G. Tao, The case for feedback control real-time scheduling, in: Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on, 1999, pp. 11–20. doi:10.1109/EMRTS.1999.777445.

[6] C. Lu, J. A. Stankovic, S. H. Son, G. Tao, Feedback control real-time scheduling: Framework, modeling, and algorithms*, Real-Time Systems 23 (1) (2002) 85–126. doi:10.1023/A:1015398403337.
URL https://doi.org/10.1023/A:1015398403337

[7] L. Abeni, L. Palopoli, G. Lipari, J. Walpole, Analysis of a reservation-based feedback scheduler, in: 23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002., 2002, pp. 71–80. doi:10.1109/REAL.2002.1181563.

[8] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, L. Abeni, Adaptive reservations in a linux environment, in: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004., 2004, pp. 238–245. doi:10.1109/RTTAS.2004.1317269.

[9] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, L. Palopoli, Qos management through adaptive reservations, Real-Time Systems 29 (2) (2005)

131–155. doi:10.1007/s11241-005-6882-0.
URL https://doi.org/10.1007/s11241-005-6882-0

[10] T. Cucinotta, L. Palopoli, L. Marzario, Stochastic feedback-based control of qos in soft real-time systems, in: 2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601), Vol. 4, 2004, pp. 3533–3538 Vol.4. doi:10.1109/CDC.2004.1429260.

[11] L. Palopoli, T. Cucinotta, Qos control for pipelines of tasks using multiple resources, IEEE Transactions on Computers 59 (2009) 416–430. doi:10.1109/TC.2009.116.
URL doi.ieeecomputersociety.org/10.1109/TC.2009.116

[12] L. Palopoli, T. Cucinotta, L. Marzario, G. Lipari, AQuoSA — adaptive quality of service architecture, Software – Practice and Experience 39 (1) (2009) 1–31. doi:http://dx.doi.org/10.1002/spe.v39:1.

[13] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, G. Lipari, On the integration of application level and resource level qos control for real-time applications, IEEE Transactions on Industrial Informatics 6 (4).

[14] T. Cucinotta, L. Abeni, L. Palopoli, G. Lipari, A robust mechanism for adaptive scheduling of multimedia applications, ACM Trans. Embed. Comput. Syst. 10 (4) (2011) 46:1–46:24. doi:10.1145/2043662.2043670.
URL http://doi.acm.org/10.1145/2043662.2043670

[15] T. Cucinotta, D. Faggioli, G. Bagnoli, Low-latency audio on linux by means of real-time scheduling, in: Proceedings of the Linux Audio Conference (LAC 2011), Maynooth, Ireland, 2011, pp. 135–142.

[16] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, pp. 4–13. doi:10.1109/REAL.1998.739726.

[17] C. S. Stangaciu, M. V. Micea, V. I. Cretu, Energy efficiency in real-time systems: A brief overview, in: 2013 IEEE 8th International Symposium

on Applied Computational Intelligence and Informatics (SACI), 2013, pp. 275–280. doi:10.1109/SACI.2013.6608981.

[18] M. Bambagini, M. Marinoni, H. Aydin, G. Buttazzo, Energy-aware scheduling for real-time systems: A survey, ACM Transactions on Embededded Compututing Systems 15 (1) (2016) 7:1–7:34. doi:10.1145/2808231.
URL http://doi.acm.org/10.1145/2808231

[19] L. Hu, W. Hu, R. Li, C. Li, Z. Zhang, A time slices based novel dvs algorithm for embedded systems, in: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, 2015, pp. 1500–1505. doi:10.1109/HPCC-CSS-ICESS.2015.163.

[20] P. Pillai, K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in: Proceeding of the 18th ACM Symposium on Operating Systems Principles, 2001.

[21] H. Aydin, R. Melhem, D. Mossé, P. Mejía-Alvarez, Power-aware scheduling for periodic real-time tasks, IEEE Transactions on Computers 53 (5) (2004) 584–600.

[22] Y. Zhu, F. Mueller, Feedback edf scheduling exploiting dynamic voltage scaling, in: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004., Toronto, Canada, 2004, pp. 84–93. doi:10.1109/RTTAS.2004.1317252.

[23] J. H. Kim, D. Gangadharan, O. Sokolosky, A. Legay, I. Lee, Extensible energy planning framework for preemptive tasks, in: 2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC), 2017, pp. 32–41. doi:10.1109/ISORC.2017.14.

[24] I. Shin, I. Lee, Compositional real-time scheduling framework, in: 25th

IEEE International Real-Time Systems Symposium, 2004, pp. 57–67. doi:10.1109/REAL.2004.15.

[25] Y. Zhu, F. Mueller, Exploiting synchronous and asynchronous dvs for feedback edf scheduling on an embedded platform, ACM Trans. Embed. Comput. Syst. 7 (1) (2007) 3:1–3:26. doi:10.1145/1324969.1324972.
URL http://doi.acm.org/10.1145/1324969.1324972

[26] N. Almoosa, W. Song, Y. Wardi, S. Yalamanchili, A power capping controller for multicore processors, in: 2012 American Control Conference (ACC), 2012, pp. 4709–4714. doi:10.1109/ACC.2012.6314995.

[27] Z. Zhang, X. Chen, D.-j. Qian, C. Hu, Dynamic voltage scaling for real-time systems with system workload analysis, IEICE transactions on electronics 93 (3) (2010) 399–406.

[28] J. R. Lorch, A. J. Smith, Improving dynamic voltage scaling algorithms with pace, in: In Proceedings of the ACM SIGMETRICS 2001 Conference, Cambridge, MA, 2001, pp. 50–61.

[29] J. Pouwelse, K. Langendoen, H. Sips, Energy priority scheduling for variable voltage processors, in: Int. Symposium on Low Power Electronics and Design (ISLPED), 2001, pp. 28–33.

[30] J. Pouwelse, K. Langendoen, H. Sips, Dynamic voltage scaling on a low-power microprocessor, in: 7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom), 2001, pp. 251–259.

[31] P. Kumar, M. Srivastava, Predictive strategies for low-power rtos scheduling, in: Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors (ICCD '00), Austin, Texas, USA, 2000, pp. 343–348.

[32] A. Qadi, S. Goddard, S. Farritor, A dynamic voltage scaling algorithm for sporadic tasks, in: Proceedings of the 24th Real-Time Systems Symposium, Cancun, Mexico, 2003, pp. 52 – 62.

[33] W. Yuan, K. Nahrstedt, Energy-efficient soft real-time cpu scheduling for mobile multimedia systems, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, ACM, New York, NY, USA, 2003, pp. 149–163. doi:10.1145/945445.945460.
URL http://doi.acm.org/10.1145/945445.945460

[34] T. Cucinotta, F. Checconi, L. Abeni, L. Palopoli, Adaptive real-time scheduling for legacy multimedia applications, ACM Trans. Embed. Comput. Syst. – Special Section on Embedded Systems for Real-Time Multimedia 11 (4) (2013) 86:1–86:23. doi:10.1145/2362336.2362353.
URL http://doi.acm.org/10.1145/2362336.2362353

[35] R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek, A resource allocation model for qos management, in: Proceedings Real-Time Systems Symposium, 1997, pp. 298–307. doi:10.1109/REAL.1997.641291.

[36] S. Ghosh, R. Raj Rajkumar, J. Hansen, J. Lehoczky, Integrated qos-aware resource management and scheduling with multi-resource constraints, Real-Time Systems 33 (1) (2006) 7–46. doi:10.1007/s11241-006-6881-0.
URL https://doi.org/10.1007/s11241-006-6881-0

[37] C. Scordino, G. Lipari, A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks, IEEE Transactions on Computers 55 (12) (2006) 1509–1522. doi:10.1109/TC.2006.190.

[38] L. Abeni, G. Lipari, A. Parri, Y. Sun, Multicore cpu reclaiming: Parallel or sequential?, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 1877–1884. doi:10.1145/2851613.2851743.
URL http://doi.acm.org/10.1145/2851613.2851743

[39] C. Scordino, L. Abeni, J. Lelli, Energy-aware real-time scheduling in the linux kernel, in: Proceedings of the ACM Symposium on Applied Computing (SAC), ACM, Pau, France, 2018.

[40] C. Scordino, L. Abeni, J. Lelli, Real-time and energy efficiency in linux: Theory and practice, ACM SIGAPP Applied Computing Review 18 (4) (2018) 18–30.

[41] J. Levin, Android Internals - Volume I: A Confectioner's Cookbook, Jonathan Levin, 2014.
URL https://books.google.it/books?id=onhDnwEACAAJ

[42] I. Kalkov, A. Gurghian, S. Kowalewski, Priority inheritance during remote procedure calls in real-time android using extended binder framework, in: Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '15, ACM, New York, NY, USA, 2015, pp. 5:1–5:10. doi:10.1145/2822304.2822311.
URL http://doi.acm.org/10.1145/2822304.2822311

[43] I. Kalkov, D. Franke, J. F. Schommer, S. Kowalewski, A real-time extension to the android platform, in: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '12, ACM, New York, NY, USA, 2012, pp. 105–114. doi:10.1145/2388936.2388955.
URL http://doi.acm.org/10.1145/2388936.2388955

[44] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, L. Ziarek, Real-time android with rtdroid, in: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14, ACM, New York, NY, USA, 2014, pp. 273–286. doi:10.1145/2594368.2594381.
URL http://doi.acm.org/10.1145/2594368.2594381

[45] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, L. Ziarek, Rtdroid: A design for real-time android, IEEE Transactions on Mobile Computing 15 (10) (2016) 2564–2584. doi:10.1109/TMC.2015.2499187.

[46] Y. Yan, K. Dantu, S. Y. Ko, J. Vitek, L. Ziarek, Making android run on time, in: 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017, pp. 25–36. doi:10.1109/RTAS.2017.38.

[47] Standard for Information Technology – Portable Operating System Interface (POSIX) – System Interfaces. IEEE 1003.1, 2004 Edition, The Open Group, 2004.

[48] L. A. Torrey, J. Coleman, B. P. Miller, A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler, Softw. Pract. Exper. 37 (4) (2007) 347–364. doi:10.1002/spe.v37:4.
URL http://dx.doi.org/10.1002/spe.v37:4

[49] C. L. Liu, J. Layland, Scheduling alghorithms for multiprogramming in a hard real-time environment, Journal of the ACM 20 (1).

[50] P. Valente, G. Lipari, An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors, in: Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International, IEEE, 2005, pp. 10–pp.

[51] U. M. C. Devi, J. H. Anderson, Tardiness bounds under global edf scheduling on a multiprocessor, in: 26th IEEE International Real-Time Systems Symposium (RTSS'05), 2005.

[52] V. Mulukutla, sched: Introduce Window Assisted Load Tracking (Oct 2016).
URL https://lwn.net/Articles/704903/

[53] J. Corbet, Per-entity load tracking (Jan 2013).
URL https://lwn.net/Articles/531853/

[54] I. Lin, B. Jeff, I. Rickard, Arm platform for performance and power efficiency — hardware and software perspectives, in: 2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 2016, pp. 1–5. doi:10.1109/VLSI-DAT.2016.7482541.

[55] A. Balsini, L. Pannocchi, T. Cucinotta, Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures, in: Proceedings of the International Workshop on Embedded Operating Systems (EWILI 2018), Torino, Italy, 2018, pp. –.