UNIVERSITY OF CALIFORNIA SAN DIEGO

Design and Optimization of the Open Speech Platform to Democratize Hearing Aid Research

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Dhiman Sengupta

Committee in charge:

Professor Rajesh Gupta, Chair
Professor Pat Pannuto
Professor Bhaskar Rao
Professor Aaron Shalev
Professor Deian Stefan

2022

The dissertation of Dhiman Sengupta is approved, and it is acceptable in quality and form for publication on microfilm and electronically

University of California San Diego

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am fortunate to have such a supporting family. My mom Sohini Sengupta, my dad Kaustav Sengupta, my sister Srijoni Sengupta, and my wife Gokce Sarar, thank you for your support and love throughout my life and during my Ph.D. Without all of you, I would not be where I am today.

I want to thank Professor Rajesh Gupta for his support as the chair of my committee, for giving me valuable guidance over the past almost six years, and for allowing me to explore different ideas of mine. Thank you for supporting me even when I struggled. It enabled me to learn so much from my time at UCSD.

I want to acknowledge Dr. Harinath Garudadri and Professor Arthur Boothroyd for their guidance and teachings about everything related to hearing aids. I have learned much from both of them, without whom my research would have no doubt taken much longer. It is their support that helped me immensely.

I want to acknowledge Professor Pat Pannuto, Professor Bhaskar Rao, Professor Aaron Shalev, and Professor Deian Stefan for being part of my committee and reviewing my dissertation.

I want to thank all of the colleagues in THELab that have contributed to the Open Speech Platform project and have made it the success it is currently. Especially the Principal Investigators of this project, Dr. Harinath Garudadri, Professor Rajesh Gupta, Dr. Ganz Chockalingam, Professor Patrick Mercier, and Professor Bhaskar Rao, for all their help and support on this project. Please visit http://openspeechplatform.ucsd.edu for the list of all people that worked with us on the Open Speech Platform.

I want to acknowledge Dr. Arthur Boothroyd's team at San Diego State University: Prof. Carol Mackersie, Paul Grand, Dr. Christine Kirsch, Dannielle Martin, Robert Novak, Shaelyn Painter, and Elena Shur for helping gather the very vital feedback from the users during their pilot field study about the Open Speech Platform. This crucial information helped shape how I designed and developed the Open Speech Platform's software framework.

I want to thank all my friends who have been my support system throughout my Ph.D. at UCSD. I especially want to thank Janet, Shravan, and Nishant for everything they did for me. They were always there to help me, no questions asked, and I am very thankful for that.

Working alongside many talented colleagues in the MESL lab has been a great pleasure. I have learned so much from each one of them.

Lastly, I must thank everyone who takes care of Chez Bob. Chez Bob was my playground where I could take my mental recess throughout my Ph.D. and was an integral part of my time at UCSD. It also fuels the entire CSE department during those lonely late nights.

Chapter 3, in part, is a reprint of the material as it appears in ”Open speech platform: Democratizing hearing aid research.” In Proceedings of the 14th EAI International Conference on Pervasive Computing Technologies for Healthcare, pp. 223-233. 2020. Dhiman Sengupta; Tamara Zubatiy; Sean K. Hamilton; Arthur Boothroyd; Cagri Yalcin; Dezhi Hong; Rajesh Gupta; and Harinath Garudadri. The dissertation/thesis author was the primary investigator and author of this paper.

VITA

2013	B. S. E. in Computer Engineering, University of Michigan, Ann Arbor
2013-2016	Computer Engineer, Naval Research Lab, Washington D.C.
2019	M. S. in Computer Science (Computer Engineering), University of California San Diego
2019	Co-Op Engineer, AMD, San Diego
2021	Graduate Teaching Assistant, University of California San Diego
2016-2022	Graduate Research Assistant, University of California San Diego
2022	Ph. D. in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Jim Yen and Dhiman Sengupta. Long distance time transfer using time reversal (t3r). In *Proceedings of the 47th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 99–106, 2016

Harinath Garudadri, Arthur Boothroyd, Ching-Hua Lee, Swaroop Gadiyaram, Justyn Bell, Dhiman Sengupta, Sean Hamilton, Krishna Chaithanya Vastare, Rajesh Gupta, and Bhaskar D Rao. A realtime, open-source speech-processing platform for research in hearing loss compensation. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 1900–1904. IEEE, 2017

Sean Hamilton, Dhiman Sengupta, and Rajesh Gupta. Introducing automatic time stamping (ats) with a reference implementation in swift. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 138–141. IEEE, 2018

Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, and Yuvraj Agarwal. Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation. In *Proceedings of the 5th Conference on Systems for Built Environments*, pages 11–20, 2018

Louis Pisha, Sean Hamilton, Dhiman Sengupta, Ching-Hua Lee, Krishna Chaithanya Vastare, Tamara Zubatiy, Sergio Luna, Cagri Yalcin, Alex Grant, Rajesh Gupta, et al. A wearable platform for research in augmented hearing. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 223–227. IEEE, 2018

Dhiman Sengupta, Tamara Zubatiy, Sean K. Hamilton, Arthur Boothroyd, Cagri Yalcin, Dezhi Hong, Rajesh Gupta, and Harinath Garudadri. Open speech platform: Democratizing hearing aid research. In *Proceedings of the 14th EAI InternationalConference on Pervasive Computing Technologies for Healthcare*, 2020

Francesco Fraternali, Bharathan Balaji, Dhiman Sengupta, Dezhi Hong, and Rajesh K Gupta. Ember: energy management of batteryless event detection sensors with deep reinforcement learning. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 503–516, 2020

Alice Sokolova, Dhiman Sengupta, Kuan-Lin Chen, Rajesh Gupta, Baris Aksanli, Fredric Harris, and Harinath Garudadri. Multirate audiometric filter bank for hearing aid devices. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, pages 1436–1442. IEEE, 2021

Alice Sokolova, Dhiman Sengupta, Martin Hunt, Rajesh Gupta, Baris Aksanli, Fredric Harris, and Harinath Garudadri. Real-time multirate multiband amplification for hearing aids. *IEEE Access*, 2022

ABSTRACT OF THE DISSERTATION

Design and Optimization of the Open Speech Platform to Democratize Hearing Aid Research

by

Dhiman Sengupta

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2022

Professor Rajesh Gupta, Chair

Hearing connects us to people. Hearing impairment has a direct impact on quality of life. However, correcting for hearing is a much more complex problem when compared to our other senses, like sight, because our brain interprets sound through a complex set of mechanisms, each with its unique functionality. Hearing aids try to compensate for many of these mechanisms and have difficulty replicating their functionality. Many researchers are actively working on designing the next generation of algorithms to address the problem of hearing loss through hearing aids. These studies are limited in their impact because of custom or commercial platforms used by different groups. Without an accessible open platform and necessary experimental infrastructure and tools, researchers are hindered in their ability to understand the algorithm's efficacy in the field and prevent adoption into widespread use. In order to address this challenge, this research presents how the Open Speech Platform (OSP), a hardware and software research tool, was designed and developed to address this gap in the hearing aid research community.

First, this dissertation shows that mobile computing platforms are suitable hardware devices for developing and testing hearing aid algorithms in the field. Here we build a proof of concept device on the Qualcomm 410c platform that we both objectively and subjectively show that it meets current best practices set by the industry.

Second, this dissertation outlines important design choices and their impact on the real-time master hearing aid (RT-MHA) framework, the software at the core of the OSP. A significant challenge in designing the framework for mobile computing platforms is dealing with a mixed real-time system on a multi-core system while sacrificing the least regarding programmability and power. This dissertation starts the design process at the system's core, a.k.a. the operating environment, and then describes how best to balance the workload across the resources available in a multi-core mobile computing platform.

Finally, this dissertation describes and evaluates the two mechanisms we designed to address the two significant challenges in developing hearing aid algorithms using the OSP. The first is the scalability of the OSP. Initially, each algorithm needs to be statically compiled, making changing algorithms difficult. Therefore, we developed a hot-swapping mechanism plus a Rapid API system to address this problem. The second is the lack of debugging after deployment. We developed Live OScope, a debugger for the OSP modeled after the oscilloscope. This dissertation shows that the Live OScope is a multipurpose tool for research.

1 Introduction

1.1 Motivation

Hearing is vital in how we interact with each other and socialize. When a person's hearing is impaired, it profoundly impacts how the person interacts with one's surroundings. Inadequate measures to mitigate hearing deficiencies often lead to isolation from the world [kSLB$^+$16], resulting in substantial economic and societal costs [O$^+$17]. In 2015, 28.8 million people in the United States suffered from hearing loss, but only 1 in 4 affected use hearing aids [NIH17]. Even though hearing aids have been shown to benefit users with hearing loss immensely [VSS$^+$17], the lack of adoption is due to self-reported hearing performance, technology commitment, and the socioeconomic and health status of the hearing-impaired persons [TGM$^+$18]. Researchers are developing novel ways to improve hearing aids in order to increase the adoption of hearing aids.

In 2014 the National Institute on Deafness and Other Communication Disorders (NIDCD) organized a workshop to incubate research that goes "beyond what is widely done today" and to identify barriers to commercializing academic research [MD14a]. The workshop found that audiologists and researchers lack the tools to quickly and cheaply test new ideas, making it hard to know how their research will translate to everyday use.

Designing hearing aids require a significant interdisciplinary effort. From a software standpoint, it requires experts from four different disciplines. The first is the signal processing expert who designs the individual algorithms, such as the filters, feedback cancellation, beamforming, and more. Next, we have hearing aid researchers who incorporate these algorithms into a hearing aid processing chain. Afterward, we need the embedded systems expert to fit the hearing aid correction algorithms into a very resource-limited hearing aid device. Finally, we have the audiologist\ clinical researcher who works with patients to understand the impacts of the algorithm in the field and study how to make hearing aids more intuitive for the end-user.

The status quo of hearing aid research faces a few key barriers. First, despite much innovative research on improving hearing aids [Kat05, KVdBMW07, DGM$^+$08, KLHL09, Kat19], the produced results and artifacts are hard to compare and reproduce [RAK$^+$19], mainly due to the lack of standardized research tools for hearing aid development. Second, major manufacturers' disparate hearing aid platforms contain proprietary, closed-source software. These barriers make it difficult for audiologists to understand the details of algorithms on board when interpreting the results produced by these devices. Furthermore, the interfaces accompanying hearing aids today enable researchers and end-users to control features such as volume but lack the flexibility that researchers need to test novel techniques [MAF19]. The lack of proper amenable research tools restricts the use of these devices, affects users' experiences, and prolongs the cycles of experiments and studies for audiologists [VV10].

The gap from algorithms to prototyping capabilities on an accessible platform prevents hearing aid designs from being tested in the field. This issue is driven by the lack of an open-sourced research system that meets the industry standard and allows researchers to test their ideas outside the lab. Until recently, due to the system's limited resources, it would require effort to take new hearing aid designs and implement them in an embedded system, which is the minimum required for field studies. However, recent advancements in the mobile computing space can make this transition from the lab to the field much faster and easier. In order to address these challenges, we designed, optimized, and implemented the Open Speech Platform (OSP), a hardware and software solution to bridge the gap. OSP is an open-source, portable, and extensible platform for hearing healthcare research. The main goal of such a platform is to give researchers a tool that easily facilitates the transfer of knowledge and artifacts between researchers in multiple domains.

1.2 Objective

This dissertation demonstrates the architectural design of a mobile computing platform suitable for hearing aid research, designs and evaluates a software framework for creating hearing aid algorithms, and defines the design flow for implementing and testing hearing aid designs for research purposes on these mobile platforms.

1.3 Technical Challenges

Hearing aids are real-time embedded systems because they must take action within specific time intervals to deliver intended functionality. Missing timing deadlines would introduce an unacceptable loss of quality. Therefore, the most significant technical challenge we address in this dissertation while building the software framework is how to ensure real-time performance by making the trade-off between power and computing ability. Another technical challenge relating to the software framework is creating a modular, systematic architecture with standard functionalities required for hearing aid research. The final technical challenge in this dissertation is designing and building portable hardware with clinical-grade behind-the-ear receiver-in-canal (BTE-RIC) hearing aids using commercial-of-the-shelf components with custom printed circuit boards and plastics.

1.4 Contributions

This dissertation makes five major contributions toward designing, implementing and optimizing a research platform for hearing aid research.

1. The open-source audio processing tools for the hearing aid research community have been looking for the best hardware platform for the next generation of hearing aid research tools. For that reason, we objectively and subjectively demonstrated that mobile computing platforms are suitable for hearing aid research by designing and developing a proof of concept platform for OSP. The platform consists of a hearing aid device built on the Qualcomm 410c platform and the foundation of a real-time software framework used to develop hearing aid algorithms. (Chapter 3)

2. As with any computing platform, the operating environment significantly impacts the system's performance, especially regarding real-time performance. Therefore, we identify the best way to set up the operating environment to get the best real-time performance for running hearing aid algorithms on these mobile computing platforms. (Chapter 4)

3. Most modern mobile computing devices contain CPUs that contain multiple efficient cores. We designed and evaluated different methods to distribute the workload across the available resource on these computing platforms to improve computation speed while trying to find the right balance between real-time performance and power consumption. (Chapter 4)

4. In order to extend the framework's capability for research, we optimized the framework's scalability further to allow for hot-swapping between multiple hearing aid designs. This optimization consisted of designing a selector mechanism for the hearing aid algorithms and designing a standardized API to interface them to external applications used to control them. (Chapter 5)

5. One of the most helpful tools for any researcher allows the researcher to track and monitor the operations of the hearing aid in-situ. Hearing aid researchers have lacked the ability to track and monitor the hearing aid's algorithm when deployed in the field. For that reason, we designed a lightweight debugging tool called Live OScope to allow the researcher to observe any part of the hearing aid algorithm in-situ. (Chapter 5)

2 Background & Related Works

2.1 Overview

In this chapter, we explore the problem space of the hearing aid community with the goal of understanding design requirements for the envisioned hearing aid platform and tools. We describe the parts of the human ear responsible for translating sound waves into neural signals that our brains can interpret. Next, we summarize the types of hearing loss and how to measure hearing loss. Then we explain how a hearing aid functions from an algorithmic side to help treat hearing loss. Finally, we provide a short survey of the different types of hearing aid research platforms the community currently uses.

2.2 The Human Ear

The human ear is a complex sensory organ made up of many components that shape how we interpret the sounds in our environment. Figure 2.1 describes the key components that impact how we translate sound into neural signals our brain can understand. The ear is described in three subgroups of components. The first is the outer ear consisting of the pinna, the external auditory canal, and the tympanic membrane. Next, we have the middle ear, an air filled cavity bridged by the ossuclar chain. This chain is made up of the malleus, incus, and stapes. Finally, we have the inner ear consisting of the cochlea and the cell bodies and dendrites of the cochlear nerve.

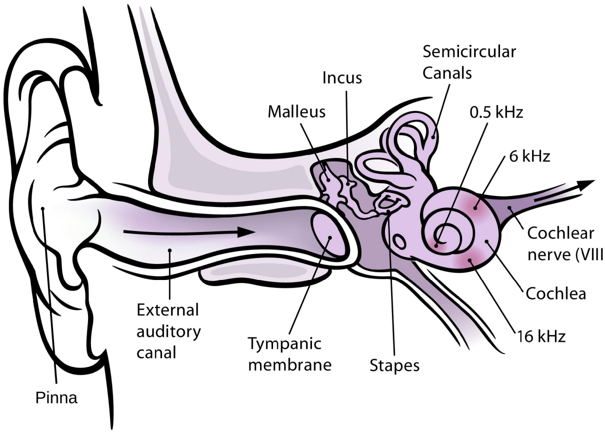


Figure 2.1: Describes the different parts of a human ear that affect how we interpret sound in our environment. This is a derived work of ”Frequency Coding in the Human Ear and Cortex” from ”Perception Space—The Final Frontier, A PLoS Biology Vol. 3, No. 4, e137 doi:10.1371/journal.pbio.0030137” by Chittka L. and Brockmann A. and is licensed under CC BY 2.5.

Each component has a unique function in converting sound into neural signals for our brains to interpret. This section summarizes the human ear at a rudimentary level, [Haw20] goes much more in-depth into this topic.

2.2.1 Outer Ear

Pinna - The pinna, also known as the auricle, is the exterior part of the ear visible to us. The pinna shapes the way we interpret sound. The shape of the pinna creates micro echos, which in combination with the way the head diffracts the incoming sound creates a complex signal pattern [YC21]. These signal patterns coming from both the ears, enables us the ability to determine the direction of a sound wave with relatively high accuracy [RHG$^+$18].

External Auditory Canal - The external auditory canal transports the sound from the pinna to the tympanic membrane. Due to the resonance in the canal, it acts as an amplifier for certain audio frequencies. Figure 2.2 shows how much amplification the canal imparts on the sound [VVG91]. Most amplification is from 2000 Hz to 4000 Hz. Not surprisingly, most of the information in a speech audio signal resides in this region [MK90]. The amount of amplification varies largely

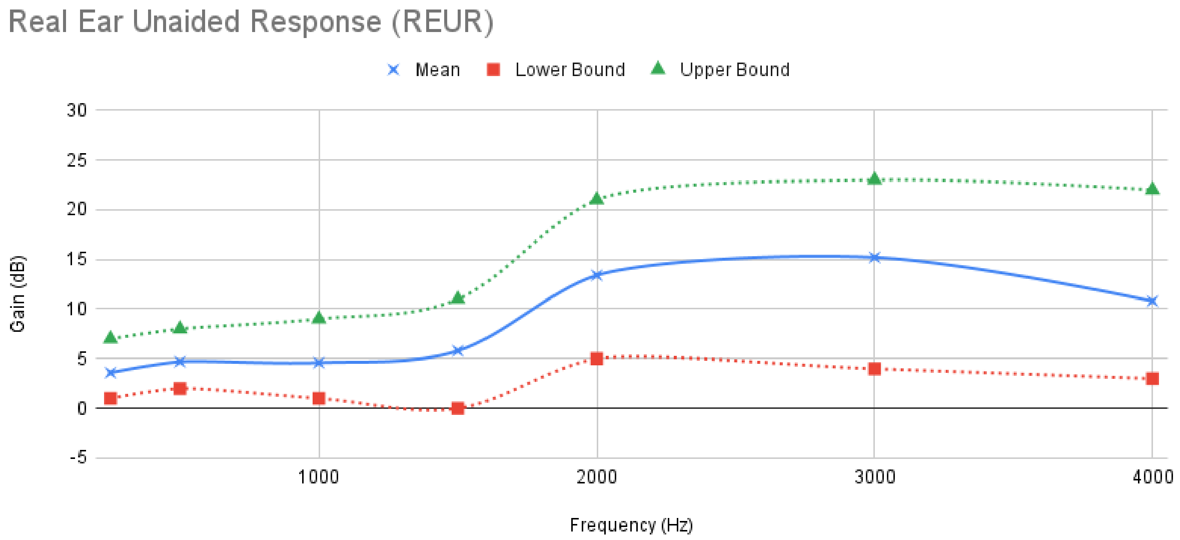


Figure 2.2: A recreation of the real ear unaided response from [VVG91], where they measured the gain from the external auditory canal, a.k.a. real ear unaided response, from 49 subjects.

between users, as shown by the upper and lower bound in Figure 2.2 and this is very important to consider when fitting a hearing aid on a person.

Tympanic Membrane - The tympanic membrane is the first link in a mechanical coupling to the cochlea, where sound is translated to signals that our brain can interpret. The size of the tympanic membrane when compared to the footplate attached to the cochlea allows for 30dB improvement in energy transfer [SG21].

2.2.2 Middle Ear

Malleus, Incus & Stapes - The primary function of these three bones is to mechanically couple the tympanic membrane to the cochlea. We require a mechanical linkage because the cochlea is filled with liquid, and without a mechanical coupling, the sound would not correctly transfer from the air to the liquid. The malleus and stapes bones are connected to two small muscles, which we control unconsciously to control the amount of sound that travels to the cochlea [Haw20].

2.2.3 Inner Ear

Cochlea & Cochlear Nerve - Once the sound wave enters the cochlea liquid, it travels through a spiral-shaped tube. The tube has three parts; scala tympani, scala vestibuli and scala media. Vibrations of the stapes generate a traveling wave along the scala media. For a single frequency, this waves increases in amplitude as it travels, reaching a maximum, after which it stops. The distance traveled before reaching the maximum depends on the frequency of vibration. In essence the scala media acts like a crude filter bank – separating complex sounds into their component frequencies. These hair cells connected to the cochlear nerve, which then transmits the the patterns of nerve impulses, through several way stations, to the auditory cortex [Haw20]. The hair cells located closer to the beginning of the tube respond best to higher frequencies, and the hair cells located towards the end of the spiral respond best to lower frequencies. The frequencies' increment decrease from the start of the tube to the end is logarithmic, meaning we have higher linear resolution at higher frequencies.

2.3 Hearing Loss

The many different components of our ear play a unique role of modifying the sound and decomposing it into its frequency components so that our brain is able to interpret the sound with all of its spatial and temporal information. If just one mechanism degrades or fails, it negatively affect our hearing. Hearing loss is categorized into three broad categories depending on which ear component is causing the issue. These categories are conductive and sensorineural.

Conductive HL - Conductive HL happens when sound waves have trouble propagating to the cochlea. This type of HL is usually due to issues in the outer andor middle ear. There are many factors that can cause this hearing loss. Some examples include blockage in the external auditory canal, a torn tympanic membrane, the middle ear bones fusing, etc. The impact of a conductive HL varies depending on the type and degree of the issue. A hearing aid can address this type of

HL as long as its within the operating range of the hearing aid. If the conductive HL is too high bone conduction hearing systems can be utilized to address the HL.

Sensorineural HL - Sensorineural HL happens when there are issues with the cochlea's ability to translate sound into signals interpretable by the brain. Some reasons for sensorineural HL are loss of sensitivity in the hair cells, damage to the hair cells, deformation to the cochlea, damage to the cochlear nerve, etc. Most commercial hearing aids are trying to address this type of HL.

Measuring HL

A pure tone audiogram (PTA) is the most common way to measure hearing loss. PTA is a test that plays tones from 125Hz to 8kHz with varying volume levels using air conduction and bone conduction. The test is used to find a person's hearing threshold for each frequency tested. Separate measures of threshold by air and bone-conduction can ascertain the relative contributions of conductive and sensorineural loss.

2.4 Hearing Aids

The hearing aid sits in front of many of these components that make up our ear. Therefore, it is important for us to understand the unique functions of each the components so that the hearing aid is able to properly compensate for each of the components impacted. The main function of the hearing aid is to map sounds helpful in interpreting the world around them into the audible range of the person wearing the hearing aid while preserving spatial and temporal cues. However, it is a balancing act as the hearing aid should not overwhelm the user by putting too much information into their limited hearing range. The mapping function is usually defined as a function of the PTA test using air conduction.

Hearing aids come in many form factors and with a wide variety of signal processing algorithms. All hearing aids have a similar foundation which consists of two major components.

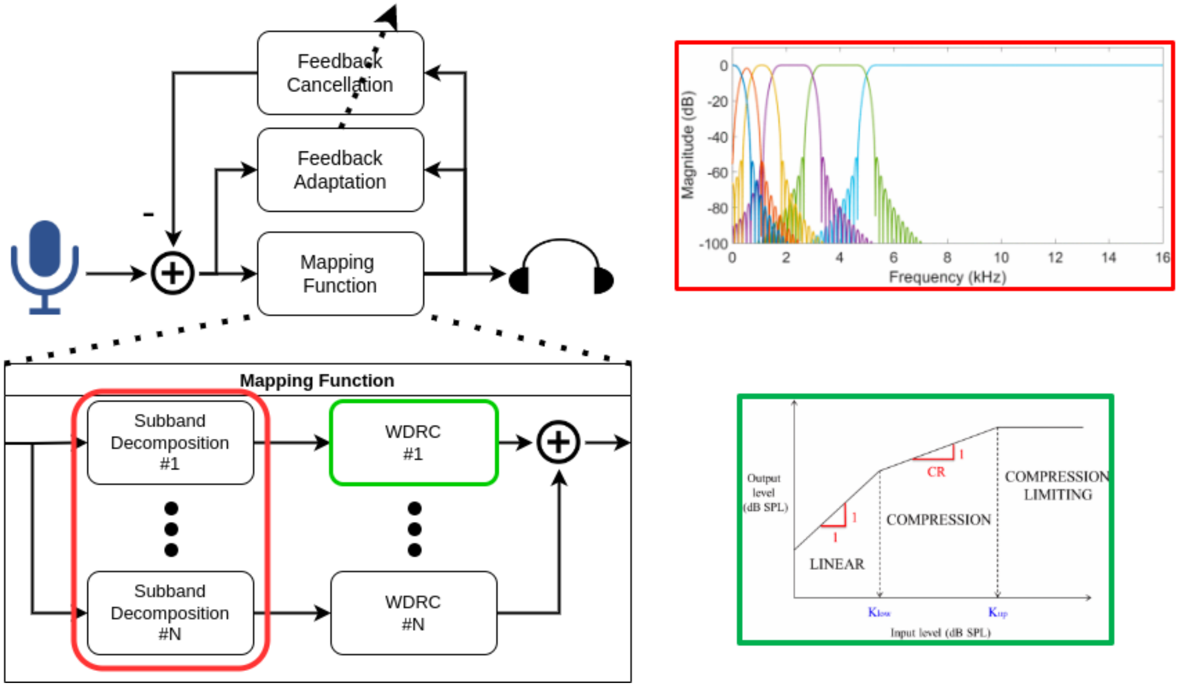


Figure 2.3: Describes a single channel basic hearing aid algorithm with its two key components. The graph in red is an example of how the hearing aid splits the incoming audio into different frequency components. The graph in green represents the WDRC mapping the sound in each frequency band.

The first component is a mapping functionality and the second is the feedback cancellation. The way a hearing aid maps audio into a person's hearing range is mainly through amplifications of the audio in discreet frequency bands. The frequency bands are usually spaced in logarithmic fashion to mimic the way cochlea interprets sound, Figure 2.3. In a basic modern hearing aid the amplification in each frequency band is done through a mapping function called a wide dynamic range compression (WDRC). The WDRC function applies gain based on the input audio level for that frequency band, see Figure 2.3. Defining the different parameters of the WDRC function is what is referred to as the prescription in a hearing aid, since it is what determines the mapping characteristic of the hearing aid.

Once an audiologist determines a person's PTA, they then run a prescription generation algorithm, like NAL-NL2, to create the parameters for the hearing aids being used. Next, the audiologist has fine tune these parameters using the person's feedback because the optimal hearing aid mapping function varys from person to person even with the same PTA. The variation can come from one of many factors related to how we percieve sound, e.g. the real ear unaided response varies quite a bit as shown in Figure 2.2. This area of how to best fit a hearing aid is a very active area in audiology, some examples are [BM17, BKM$^+$19, MBG18, MBL19, MBG20].

A hearing aid is located externally usually somewhere on the person's pinna with the speaker located at the entrance of the external auditory canal. Therefore, the sound needs to be recorded, amplified and played in a very close proximity. This environment is prone to feedback between the speaker and the microphone because of the high amplification and the close proximity. Which is why hearing aids require a basic form of feedback cancellation, usually an adaptive feedback cancellation. The reason why adaptive is preferred is due to the environment around the hearing aid changes frequently which changes the feedback path.

Even though, a basic hearing aid does help the user quite a bit, it is not perfect and have many short comings. For example, since some hearing aid bypass the pinna completely by place the microphone behind the ear, directionality of sound becomes an issue. For this reason, more advanced hearing aids build upon these two foundational components to enhance the hearing aid using advanced signal processing algorithms such as spatial cue preservation to address these short comings [JKG21]. A lot of researchers are trying to use novel signal processing techniques to further improve the hearing aid's functionality and it is a very active area of research.

Extensive research has been done in advancing hearing aids [KLHL09, DGM$^+$08, Kat08, KVdBMW07, Kat19, Kat05]. Unfortunately, it is hard to determine how effective these algorithms might be for the end user because many advanced algorithms are not tested in the field with human subjects. Even hearing aid manufactures implement advance signal processing algorithms without fully understanding the effects on their users. For example, using beamforming to suppress noise for hearing aids has been implemented in some hearing aids. However, in practice it is only usable when the person wearing the hearing aid is close to the signal they are interested in or else the algorithm starts to suppress relevant information we use to interpret the sound. One proposed reason for this issue is that after a certain distance a lot of the information we get from the signal comes from the reverberations in a room which gets suppressed by the beamforming algorithm. What this demonstrates is that there is a need to test in the real world to better understand the short comings of the algorithm. In academia the reason for the lack of this type of testing is largely due to a lack of tools available for field testing.

2.5 Related Works: Hearing Aid Research Tools

Recently the hearing aid research community has been pushing for open source tools that can be used to quickly test out their ideas in the real world. In this section we explore the different tools available and we organized them based on the hardware technology that they were designed for. Figure 2.4 visually describes this organization and based on this survey we can say with confidence that mobile platforms are the sweet spot for a research tool. Mobile platforms are the first hardware technology going from left to right with the ability to be used in field studies. This means that mobile platforms are the most computationally powerful technology with the ability to be used in real world studies. For this reason we developed OSP primarily for mobile platforms and this thesis explains the designing of the OSP framework. The rest of this section describes the other available hearing aid research tools.

2.5.1 Lab Based Systems

Currently, most research is done on lab based systems using of the shelf hardware and software. The hardware Many signal processing researchers use programming languages such as Matlab, DADiSP, or Python. These programming languages are great for prototyping algorithms and performing testing in a simulated environment. Some testing with human subjects in a lab environment is also possible using pre-recorded data. However, these languages have a high overhead cost associated with them which prevent algorithms from running with low latency,

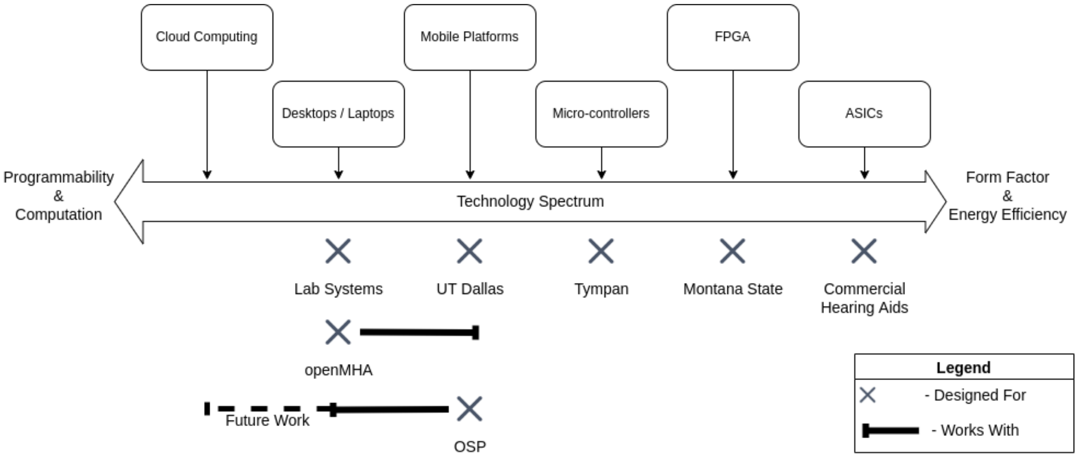


Figure 2.4: Hearing aid research tools organized based on their type of hardware technology they are meant for. The figure describes what each tool was primarily developed for and what they can work with.

which is required for real-time processing. Therefore, most frameworks that we discuss use the C++ programming language since it has a very low overhead cost and is suitable for low latency real-time applications. The other issue with current lab based systems is that there is no standardized framework making it hard to compare and reproduce results and artifacts of other's work [RAK$^+$19].

2.5.2 OpenMHA

HorTech and the University of Oldenburg released openMHA [HKL$^+$17, KHM$^+$19, KHM$^+$22] in 2017, which is a software only solution based on the closed-source commercial HorTech Master Hearing Aid software[1]. OpenMHA is a development and evaluation platform with a vast DSP library that is capable of executing hearing aid signal processing in real-time on standard computing hardware with low latency ($<$ 10 ms) between the sound input and output. The openMHA framework has two key design objectives. The first is the ability to configure the hearing aid algorithm at run-time using their own configuration language and dynamic libraries. The second design objective of openMHA framework is having the ability to run on many devices.

[1] https://www.hoertech.de/

These objectives indicate that they wanted to create a generalizable hearing aid framework. This works really well for modern day laptops and desktops given that they have plenty of computation power to handle the demands. However, this comes at a cost, such as not being able to take full advantage of multiple-core system for the signal processing which can be detrimental to the amount of computation that can be done on a mobile platform. Similarly, dynamic libraries inherently has a higher overhead compared to static libraries. So what we can gather from this is that the openMHA framework was primarily designed for desktop and laptop devices, which was then extended to mobile platforms like the Raspberry Pi and Beagle Bone Black[2].

OpenMHA allows the user to tune hearing aid parameters over a network socket using a control interface. This control interface allows researchers and developers the ability to create application on top of the openMHA framework. The control interface is not just used for controlling the parameters but the entire configuration of the hearing aid in openMHA. Which is a lot of information that can be overwhelming for researchers when trying to develop their own application. Plus, the configuration programming language for openMHA does not allow the hearing aid developer any control over how parameters are grouped or displayed.

2.5.3 UT-Dallas Smart Phone Based System

The Statistical Signal Processing Research Laboratory at University of Texas Dallas is planning on developing a smartphone-based open-source research platform for hearing improvement studies as a smartphone-app solution for Android or iOS running on various hardware [Lab20]. Currently, this platform consists of mainly a set of signal processing algorithms[3] but is missing a framework to design complete hearing aid algorithms or an interface for custom applications which are required for a research platform in this area [KP17]. One major thing of note in their study [Lab20], only the iPhone models demonstrated a latency of less than 10

[2] http://www.openmha.org/hardware/

[3] https://labs.utdallas.edu/ssprl/hearing-aid-project/research-papers/

ms, which means that end users will most likely be able to notice the delay between when audio is picked up by the mic and played back in the ear. This is because users without hearing loss can tolerate at most 10ms, especially when it comes to their voice [GCBM18]. The reason most applications running on smartphones running Android or iOS can not get lower than 10 ms is because these operating systems lack mechanisms required to optimize the operating environment for low latency tasks.

2.5.4 Tympan

Tympan developed a low-cost open-source portable hardware platform for hearing aid research and development with very intuitive software development tools [Tym20]. The hardware platform is based on a small embedded microprocessor, which is great testing platform for optimizing hearing aid algorithms to see whether it would fit in a commercial hearing aid. However, the lack of processing power in this platform would restrict the type of algorithms that can be run on such a device, such as least mean square based adaptive feedback cancellation while processing six wide dynamic range compression bands.

On top of a lack of processing power, Tympan lacks an easy to use control interface. Without such control interface it becomes relatively difficult to design and implement an external application especially for the hearing aid research community.

2.5.5 Montana State University FPGA Framework

Using an FPGA is a very cumbersome process and mainly intended to be used to test out hardware designs of a hearing aid algorithm which has already been finalized. Once the hardware has been verified on an FPGA it is sent to be produced as an application specific integrated circuit (ASIC), which is usually what we get in commercial hearing aids. The work out of Montana State University [V$^+$20] is trying to change that by building a framework which extends Matlab's

Simulink HDL code generator tool but the framework requires an FPGA with system on a chip (SoC), an SoC is something similar to a CPU on a mobile platform. The Matlab tool takes a high-level description of the hearing aid algorithm and automatically generates code for the FPGA. This work [V^{+}20] extends this tool by automatically generating a graphical user interface and it corresponding code for the FPGA with the SoC. The framework has potential to be used to develop ASICs but as hearing aid research tool the barrier to entry is too high because it still requires a lot of manual effort to implement and optimize the hearing aid algorithm for an FPGA.

2.5.6 Commercial Hearing Aids

Commercial hearing aids are usually built on a very low power ASIC, which means that there is little to no programmability available to change the hearing aid algorithm. Therefore, researchers can only test hearing aid algorithms which come with the commercial hearing aid. If the researcher want to test out any new hearing aid algorithm using a commercial hearing aid, they have to wait for the manufacturer to implement it in their product. This wait can be a very long time because of how complex it is to design and implement new algorithms in ASICs. Commercial hearing aids can be a good choice for doing field studies on algorithms already implemented. However, the researcher would be at mercy of the manufacturer if they wanted to try novel algorithms on a commercial hearing aid.

3 The Open Speech Platform

3.1 Design Philosophy

The goal of the Open Speech Platform (OSP) is to make a platform that is accessible to non-specialist, audiologist alike to prototype and experiment with different algorithmic means to improve the hearing experience. Audiologists and researchers will benefit from a research system with several essential properties [MD14b]. First, the system should be entirely open-source, both hardware and software, which allows for a standardized experimental workflow enabling researchers to evaluate and compare research results as well as artifacts. Second, the system should contain a user interface that allows researchers to bring patients into the loop, is simple to edit, and exposes the internal components and details (e.g., signal processing techniques) of the hearing aid for transparent control. Third, the system should contain a standard tool for conducting experiments ”in the wild.” Finally, the system should be able to both accurately and reliably reflect the acoustic properties of hearing aids commonly used today.

The rest of this chapter will discuss, based on this design philosophy, the design of the software, the design of the hardware, and the evolution of the OSP. All software and hardware described in this paper is open-source and available at github.com/nihospr01/OpenSpeechPlatform-UCSD.

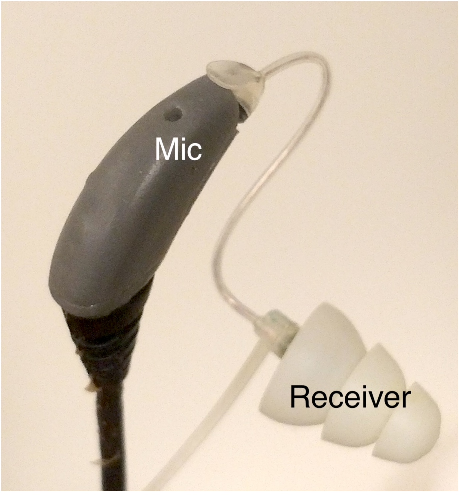


Figure 3.1: The BTE-RIC hearing aid developed for both the lab-based system and the portable system.

3.2 Hardware

This section describes the design of three hardware devices we developed: the hearing aid, the lab system, and the portable system.

Hearing Aid Design

We designed and developed a clinical-grade hearing aid, based on the most common hearing aid form factor called behind-the-ear receiver-in-canal (BTE-RIC). The design of this hearing aid is simple yet effective; it has one MEMS microphone for the input and one clinical-grade receiver for the output, as shown in Figure 3.1. Internal to the hearing aid, there is a pre-amplifier circuit, which amplifies the microphone signal before sending it along the wire in order to make the signal noise tolerant. The receiver is connected directly to the wire and requires no amplification in the hearing aid.

The challenge in developing the hearing aids, however, is designing the plastics. The shape and size of the plastic have a significant impact on the acoustic quality of the hearing aid. The cavities created by the plastic shell resonate at particular frequencies, which causes significant

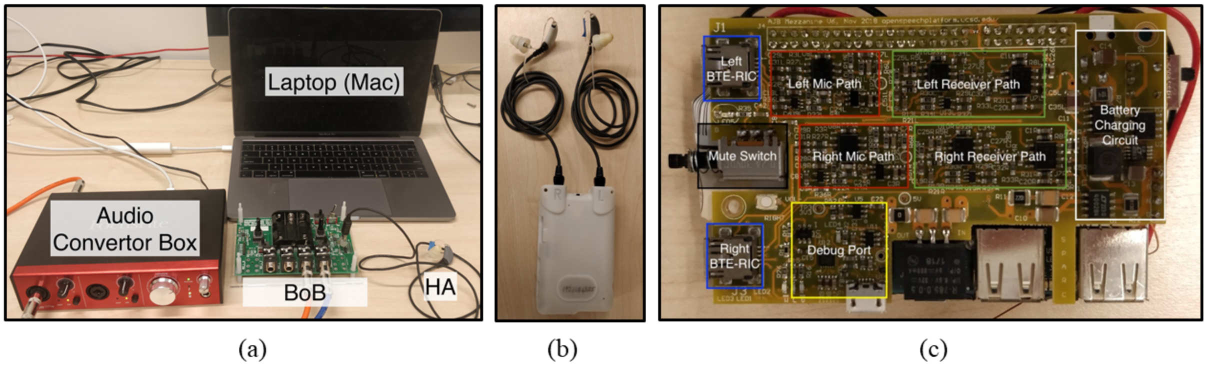


Figure 3.2: a) Lab-based system consisting of a laptop (Mac), an audio converter box, a break out board, and a hearing aid. b) The portable system connected to two hearing aids. c) Annotated PCB view of the daughterboard that attaches to the the DragonBoard 410c SBC.

feedback between the microphone and the speaker to overwhelm the Adaptive Feedback Cancellation algorithm. After understanding how the plastic shell affects the acoustics, we designed the BTE-RIC hearing aid shown in Figure 3.1, which performs on par with other commercial hearing aids, Table 3.1.

Lab System

During the development of the hearing aids, we created a lab-based system, which allows the researcher to use any computer installed with either Linux or macOS. Our lab system, Figure 3.2a, is comprised of a laptop, an off-the-shelf audio converter, a break out board (BoB), and a hearing aid we designed. The audio converter box is a Focusrite box with a Thunderbolt connection to the Mac. Finally, the BoB is a custom printed circuit board (PCB) for amplifying both the signal coming from the microphone to the Focusrite box and the signal transmitted from the Focusrite box to the receiver. The BoB ensures that signal feed into the Focusrite, and the signal that goes to the receiver is at the correct level.

Even though in our setup, we have the BoB connecting to the Focusrite box, the BoB can connect to any audio conversion box. Therefore, this system allows researchers to design and develop on any computer while having access to hardware, which is equivalent to commercial hearing aids.

Portable System

In order to enable experiments in the field, we also designed a portable version of the system , Figure 3.2b, which is composed of a DragonBoard 410c, a single board computer (SBC), a custom printed circuit board (PCB) daughterboard, and a 23 Wh rechargeable battery. All of these are packaged into a 3D printed shell, which we envision users would hang around their neck while using the hearing aids.

The first step in designing the portable system was choosing the embedded computer, which would be the foundation of the portable system. We decided on the DragonBoard 410c SBC because it achieves a balanced trade-off between computational power and power efficiency, with an active hobbyist community to support its use and development. The DragonBoard 410c consists of a quad-core ARM A53 chipset with 1 GB of RAM and 8 GB of memory. The board contains WiFi, Bluetooth, and GPS. The DragonBoard 410c also has a built-in audio codec.

In order for the hearing aid to interface with the codec, we needed to design a daughterboard with similar functionalities as BoB from the lab-based system. Therefore, we designed a PCB, Figure 3.2c, that interfaces with the DragonBoard 410c SBC, which amplifies the signal from the microphone and to the receiver. The board also contains a mute switch for safety measures and a debug port allowing researchers to access the portable device over USB. Lastly, the daughterboard can recharge and operate from a 23 Wh battery, which gives the system around 12-hour battery life.

Lastly, the portable system runs Debian Linux, which makes porting code between the lab system and the portable system relatively easy. This means that we can develop on the lab-based system and then test on the portable system when the code is ready.

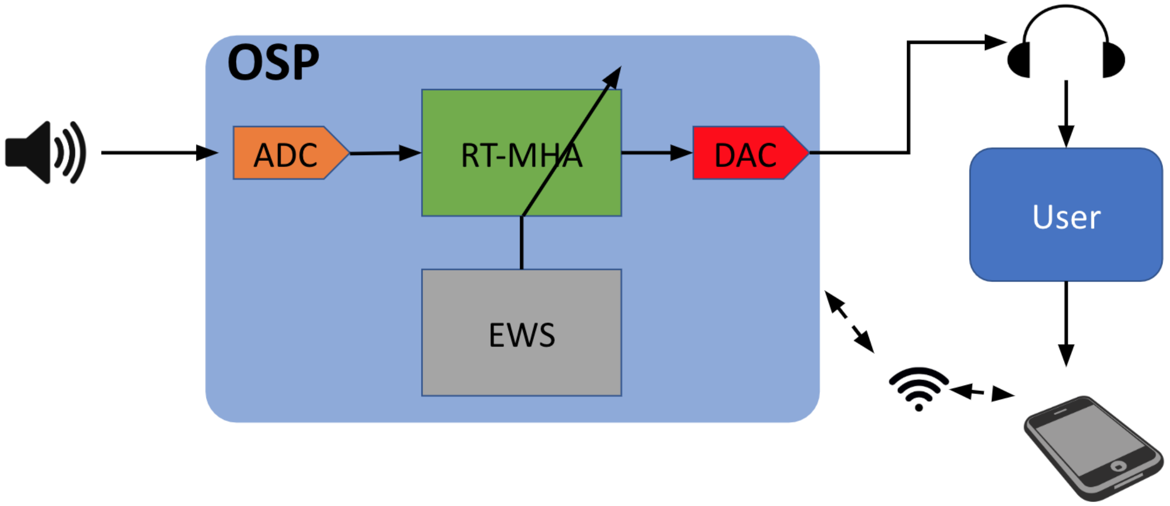


Figure 3.3: Overview of the Open Speech Platform (OSP): OSP consists of both open-sourced hardware that meets the industry standards and software equipped with open, modular architecture and amenable user interfaces as well as functionality.

3.3 Software

At the core of OSP software, Figure 3.3, is the real-time master hearing aid (RT-MHA), which is in charge modifying the audio stream in real-time. At the same time RT-MHA needs to change the behavior of the audio processing when it receives commands from the external control interfaces, such as the embedded web server (EWS). The EWS hosts a suite of web apps which allow the user to control RT-MHA through graphical user interfaces (GUIs). We further elaborate on the design of each of the components.

RT-MHA

When designing the RT-MHA framework, our goal is to create a modular environment that abstracts the details of the real-time hearing aid algorithm (i.e., common functionalities such as wide dynamic range compression (WDRC), speech enhancement, feedback cancellation, etc.) into simple, extensible, and user-friendly application programming interfaces (APIs) . As

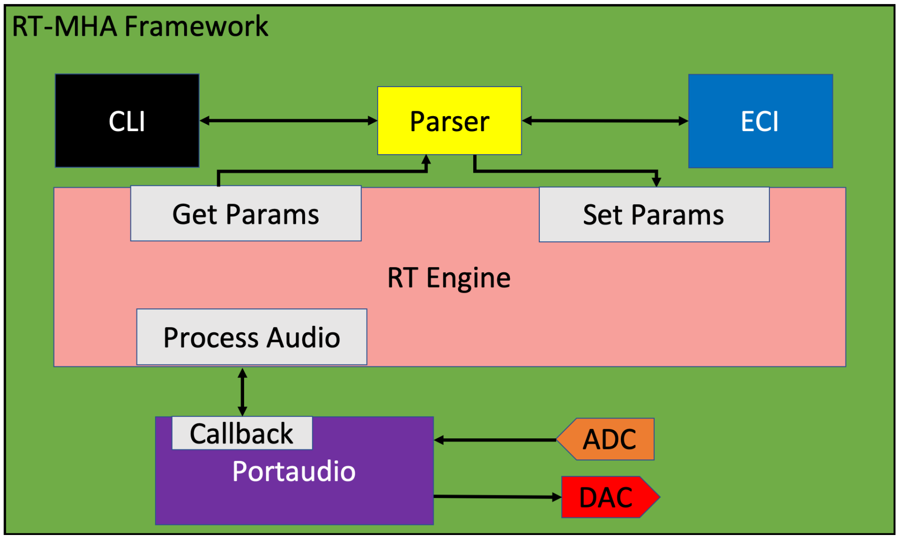


Figure 3.4: Architecture of real-time master hearing aid.

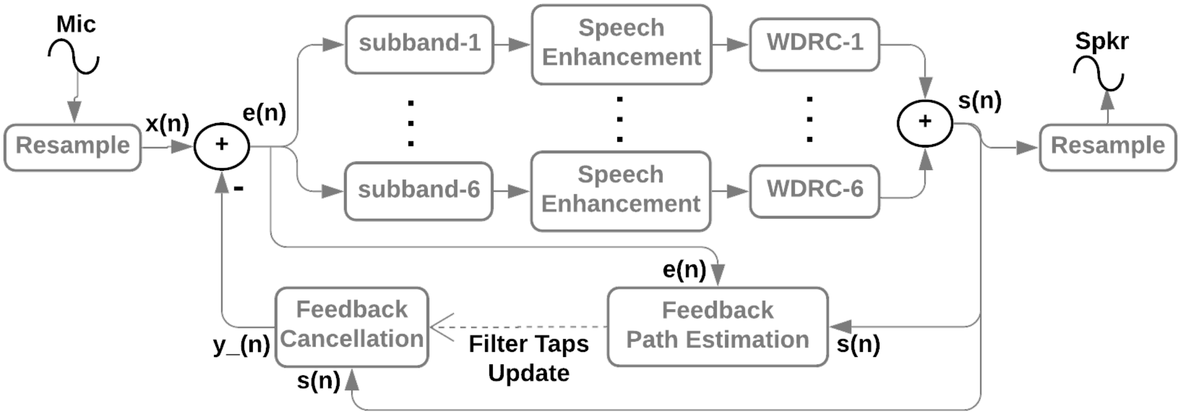


Figure 3.5: Digital signal processing path for the reference RT-MHA implementation, comprised of audio resampling, sub-band filtering, speech enhancement, wide dynamic range compression, and feedback path estimation as well as feedback cancellation.

illustrated in Figure 3.4, RT-MHA hosts a few key components: 1) a central Real-Time Engine that handles all the callbacks issued by PortAudio and orchestrates the processing of the audio; 2) an auxiliary Parser interface to parse the input from users through the programmable or graphical interfaces and set the parameters of modules and algorithms in the hearing aid; 3) a external connection interface for communicating with GUIs; 4) a command line interface (CLI) tool for debugging purposes; 5) a PortAudio module that issues callbacks for further processing when audio data arrive.

Real-Time Engine The core of the RT-MHA framework is the Real-Time (RT) engine, a C++ development environment that provides the developers with a real-time modular environment to develop their hearing aid algorithms. We envision that implementing hearing aid algorithms in the RT engine would be as simple as to compose a structured ensemble of library modules. An example of this is the reference hearing aid algorithm that we provide as a part of the OSP system, as illustrated in Figure 3.5. This reference design is based on Kates work [Kat08] and incorporates the fundamental algorithms that are necessary for hearing aid. The following briefly describes the function of each of the components in the reference design, for more details see [GBL^{+}17] :

- *Resample 3:2 and 2:3*: These blocks re-sample the input stream from 48 kHz to an output at 32 kHz and vice-versa.
- *sub-band-N*: This block is a band-pass filter that separates the incoming signal into one of N different sub-bands for further processing (6 bands in the reference design).
- *Speech Enhancement*: This block uses statistical data analysis to detect the presence of background noise and suppresses it from the output signal.
- *WDRC-N*: The Wide Dynamic Range Compression (WDRC) block takes the input signal and a sub-band and remaps the signal from full scale to a restricted range.
- *Feedback Cancellation*: This block uses the estimated transmission path between the BTE-RIC's output speakers and microphones to reduce ringing from feedback.
- *Feedback Path Estimation*: This block periodically estimates the transmission path and updates the filters coefficients in the Feedback Cancellation block.

The key to creating a modular real-time environment starts with the basic building blocks, which, in our case, are the *library modules*. Therefore, we developed a template on how to design

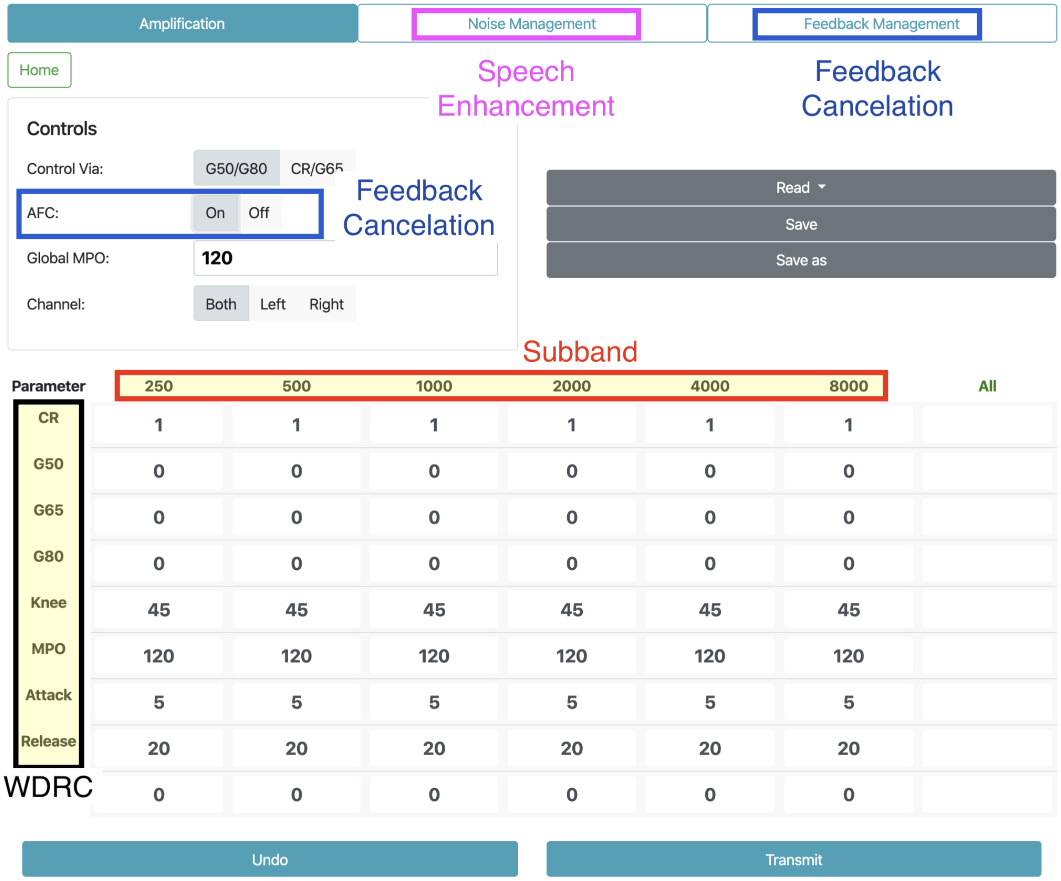
Algorithm 1 Modular library skeleton for signal processing algorithms.

```
class libModule{
public:
    ...
    /*@brief Setting libModule parameters*/
    void set_param(...){
        /* Create a new struct for the incoming param*/
        std::shared_ptr<libModule_param_t> next_param =
                std::make_shared<libModule_param_t> ();
        std::atomic_store(&globalParam, next_param);
    }
    /*@brief Getting libModule parameters*/
    void get_param(...){
        /* Load the current global param structure atomically*/
        std::shared_ptr<libModule_param_t> localParam =
                std::atomic_load(&globalParam);
        /* Return all of the parameters by reference*/
    }
    /*@brief Real-time processing inside libModule*/
    void process(...){
        /* Load the current global param structure atomically*/
        std::shared_ptr<libModule_param_t> localParam =
                std::atomic_load(&globalParam);
        /* This is where the real-time code will go*/
    }
private:
    struct libModule_param_t {
        /* Set of Parameters */
    };
    /*The pointer to the global param structure*/
    std::shared_ptr<libModule_param_t> globalParam;
};
```

a library module (see Algorithm 1). The template dictates that the library module must have a function for setting the parameters, a function for getting the parameters, and a function for processing the real-time audio data. This template requires each library to have a private data structure for communicating data between the functions using a global shared pointer. The global shared pointer points to the currently valid data structure, and only the set parameter function can update the global shared pointer. When the set parameter function, for example, needs to update the parameters, it creates a new data structure. It then atomically replaces the global pointer to point to the new data structure. As for the other two functions, they can only atomically load the data structures and read the values contained in them. This method of one-writer-multiple-readers allows us to create a lock-free environment with minimal interaction between the functions, which ensures that non-real-time interactions will not impact real-time performance. This style of coding is pervasively used in the audio industry [Dou15], and we deem it the best model when developing the OSP system. The benefit of adopting such a design is that modules developed using this template are modular and capable of uninterrupted real-time processing.

On top of providing a modular environment, the RT engine can create three domains for audio processing to spread the processing over three real-time threads. The first domain is the binaural domain, which processes the algorithms that require a signal from both ears. The second is the left domain, which deals with algorithms that modify signals for the left ear. Similarly, the right domain is for the right ear. The three domains allow for more efficient use of a multiple-core system, like our portable device in section 3.2.

Parser The parser is the gatekeeper when deciding what parameters are accessible to users through the programming or graphical interface in OSP. Exposing the internal parameters to the users would provide them with transparent control over the internal components of an algorithm. The algorithm developer will decide what parameters to expose in the user-facing interface by modifying the parser for their RT engine. Based on the list of parameters to expose, the parser



Control Via: [✓ G50/G80] CR/G65

AFC: [✓ On] Off

Global MPO: 120

Channel: [✓ Both] Left Right

Parameter	250	500	1000	2000	4000	8000	All
CR	1	1	1	1	1	1	
G50	0	0	0	0	0	0	
G65	0	0	0	0	0	0	
G80	0	0	0	0	0	0	
Knee	45	45	45	45	45	45	
MPO	120	120	120	120	120	120	
Attack	5	5	5	5	5	5	
Release	20	20	20	20	20	20	
	0	0	0	0	0	0	

Figure 3.6: Example graphical user interface for adjusting the reference design, mainly for researchers.

would then create a JSON string-based API that contains these parameters, which is how the services such as the EWS (which we shall explain shortly) would interact with RT-MHA.

External Control Interface (ECI) The ECI module is the process in the RT-MHA framework that listens on a TCP/IP port for incoming commands from a control application. The parser serializes the parameters into a human-readable JSON format, which the ECI module then sends to the control application when requested. When the control app needs to change the parameters, it will send a human-readable JSON string of the parameters with the new values back to the RT Engine through the parser via the ECI module. The parser would update the parameters atomically without interrupting the RT Engine. This mechanism allows for any type of GUI to interface with RTMHA.

Command Line Interface (CLI) For debugging and rapid prototyping of RT-MHA, we designed a simple command-line interface that can interact with RT-MHA through the parser. The CLI provides the developer with a quick and convenient way to test both the Parser and RT-MHA without the need to connect through a separate control application.

PortAudio In order to create a device- (and operating system) agnostic framework for real-time master hearing aid development, we need to abstract the low-level details about how the algorithms interact with the hardware. Through experimentation, we discovered that PortAudio[1], open-source audio I/O library, meets all of our design needs, like providing a comprehensive abstraction of the hardware, being open-source, supporting Linux/macOS/Windows, and allowing for low latency input/output.

EWS Framework

A common choice for providing a means for the user to interact with a system and control the components is to develop phone applications (e.g., on Android or iOS). However, for a development platform such as OSP that is intended for a broad spectrum of users from researchers to audiologists, a phone application will dictate a limited set of devices people could use to interact with RT-MHA. According to Xanthopoulos et al. [XX13], web apps are becoming an increasingly better option for mobile app development because there are many libraries, such as HTML5 and JQUERY, which simulate native app functionality. Plus, the development time is much shorter and less complicated than for native apps. Therefore, we choose to build an environment that would allow us to create web apps — browser-based applications downloadable through the web. This way, these web apps can be used from any browser-enabled device.

[1] http://www.portaudio.com

To this end, we have incorporated an Embedded Web Server (EWS) into our OSP platform, which hosts a suite of web apps, like the researcher page in Figure 3.6. These web apps are responsive to the form factor of the device. The responsive web apps are platform-agnostic [SHG13] allowing users to control the RT-MHA from any web-enabled device.

EWS is based on modified Linux-Apache-MySQL-PHP (LAMP) architecture. The Apache server has been replaced with a light-weight HTTP server provided by the Laravel framework. MySQL has been replaced by SQLite for simplicity and to minimize resource usage.

The web-apps are implemented primarily in HTML5, CSS, and JavaScript. This way, incorporating web-apps reduces the barrier to creating tools to control the RT-MHA for hearing aid application developers.

3.4 Evaluation

In this section, we empirically evaluate the usability of the OSP platform concerning three different criteria. First, we evaluate the hardware in terms of how well it corrects for the end users' hearing loss. Then we determine whether the latency caused by the audio processing by the platform meets the requirements set forward by the series of studies called the Tolerable Hearing-Aid Delays [SM99, SM02, SM03, SM05, SMMD08]. Finally, we evaluate the impacts of the user interacting with the platform on the real-time performance of RT-MHA.

3.4.1 Hearing Hardware Evaluation

We first examine the quality of the hearing aid hardware developed for both the lab-based system and the portable system by comparing them against four commercially available hearing aids.

Table 3.1: ANSI 3.22 test results for OSP system configurations measured by Audioscan Verifit 2, as compared to results from four commercial HAs.

Metric	*Units*	**Commercial**				**Lab**		**Portable**		**ANSI**
		A	**B**	**C**	**D**	**X**	**Y**	**X**	**Y**	**3.22**
Average Gain @ 60 dB	*SPL*	40	40	25	35	40	40	35	39	–
Max OSPL90	*SPL*	107	112	110	111	121	130	119	130	–
Average OSPL90	*SPL*	106	109	108	106	112	126	111	126	–
Average Gain @ 50 dB	*SPL*	37	39	25	35	35	41	35	40	–
Low Cutoff	*kHz*	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	≤ 0.2
High Cutoff	*kHz*	5	6	5	6.73	8	6.3	8	5	≥ 5
Equivalent Input Noise	*SPL*	27	26	30	27	29	28	38	39	≤ 30
Distortion @ 500 Hz	*% THD*	1	1	0	0	2	1	1	3	≤ 3
Distortion @ 800 Hz	*% THD*	1	1	0	0	3	2	1	2	≤ 3
Distortion @ 1600 Hz	*% THD*	0	0	0	0	1	1	1	1	≤ 3

X: with high-bandwidth Knowles receiver [Ele09]
Y: with high-power Knowles receiver [Ele14]

Setup

To test the quality of the hearing aids, we use a test created by the American National Standards called ANSI 3.22-2003 [Jor16]. The test mainly is used to verify the claims made by the manufacturer of the hearing aid. Therefore, we compared our lab and portable setup against four commercially available hearing aids.

In order to perform this test, we used the Verifit 2 test equipment by AudioScan[2]. It isolates the hearing aids in a quiet environment and uses its calibrated speakers and microphones to measure the ten different metrics of the ANSI 3.22-2003 test. During our testing, we tested two different receivers in our systems. The first receiver, receiver X, is a high bandwidth Knowles receiver for mild to moderate hearing loss and has a higher frequency fidelity. On the other hand, receiver Y is a high power Knowles receiver that is used for users with moderate to profound hearing loss.

[2] https://www.audioscan.com/en/verifit2/

Results

The first four rows of Table Table 3.1 shows that both the systems are able to correct for the same amount of hearing loss as any of the commercial hearing aids we tested. According to the rest of Table Table 3.1 both the lab system and portable system are within specification with the only exception being the equivalent input noise for the OSP Portable System. We identified the codec on the Dragonboard 410c board as the offending hardware that causes the high input noise. The audio codec is part of the power management IC (PMIC) on the Dragonboard 410c board, and therefore the switching noise of the PMIC is picked by the microphone circuitry, which is the reason for the higher than usual noise. However, as we will see in the usability study in Section section 3.5, most hearing-impaired users were either not able to notice the noise or not too bothered by it.

This test verifies that the hearing aid we built, along with the underlying software, is close to the specifications of commercial hearing aid, promising its adoption and use in research experiments and studies.

3.4.2 Processing Latency of OSP

In a series of studies called Tolerable Hearing-Aid Delays [SM99, SM02, SM03, SM05, SMMD08], audiologist and psychologist determined that 20-30 ms is the maximum latency a user of a hearing aid can tolerate. Latency refers to the time it takes audio to be captured by the hearing aid and played back into the listeners' ears. The longer the latency, the more uncomfortable it becomes to use the hearing aid; according to these studies, at the 20-30 ms mark is when users start to notice the latency becoming unacceptable. Therefore, we need to design and ensure that the combination of the hardware and software is within the acceptable range.

Table 3.2: Latency results for both the OSP systems and the RT-MHA reference design.

	Latency (ms)
Lab System (Mac)	5
Portable System	3.2
RT-MHA Reference	3.3
Lab + RT-MHA	8.3
Portable + RT-MHA	6.5

Setup

To evaluate the latency of OSP, we consider both the hardware and software. The devices were set up in loop-back mode, where the output was directly fed into the input. Then we played a sound through the device and recorded the input of the device at the same time. Comparing the input and output files, we can determine the latency of the hardware by calculating the latency between the two files. Next, we performed the same test except for playing the sound through the RT-MHA algorithm and then recording the output, helping us determine the software attributed latency. By adding the combination of the recorded latency, we can determine the overall end-to-end latency of the device when in use.

Results

The results in Table 3.2 indicate that the reference design of RT-MHA can be run on either the lab system or the portable system and is still below the 20-30 ms latency threshold. This verifies that the current hardware and software design can be used in clinical trials. The results also show that there is plenty of resource in terms of latency for more sophisticated algorithms to be implemented in RT-MHA.

3.4.3 Impact of User Interaction on Real-Time Performance

The usability of a hearing aid system is rooted in its real-time performance. RT-MHA receives one data packet of audio every 1 ms and needs to process it before the next one arrives. If

RT-MHA misses its deadline to process the audio, the user would hear “clicks” and “pop” sound. Audio artifacts are acceptable once in a while; however, continually hearing these artifacts can make hearing aids unusable. Algorithm developers have to ensure that their software can meet real-time requirements. Developing real-time software usually involves anticipating how the interaction between the non-real-time processes will impact the real-time performance. We shall demonstrate next that the RT-MHA framework in OSP has successfully decoupled the effects of non-real-time processes on the real-time algorithm. This means that neither the UX designer nor the algorithm developer has to worry about how the software they develop will impact the other.

Setup

In this experiment, we set up the portable device with two BTE-RIC hearing aids and enabled all of the hearing aid functionalities described in the reference design. The software was instrumented with real-time profiling code, which calculates the *time* taken for RT-MHA to complete its processing. The real-time profiler allows us to determine whether the master hearing aid (MHA) algorithm can meet the soft real-time requirements using the resources available on the system. It also enables us to evaluate the effects of a user interfacing with the device on the real-time resources that RT-MHA requires. Notably, we explore the impacts of user interaction with the platform, through the EWS, on the real-time master hearing aid algorithm.

We started by running all of the algorithms enabled in the reference design and profiled the real-time resources used by RT-MHA for an hour as the baseline. Then we connected a secondary device via Wifi broadcasting from the device. The secondary device has a python script running, which randomly changes the parameters of the reference design running on the device at a particular interval to mimic a user interfacing with the device. In our evaluation, we chose three different intervals — 30 seconds, 10 seconds, and 1 second, ran the experiment for an hour for each interval, and measure the execution as mentioned earlier to evaluate the performance.

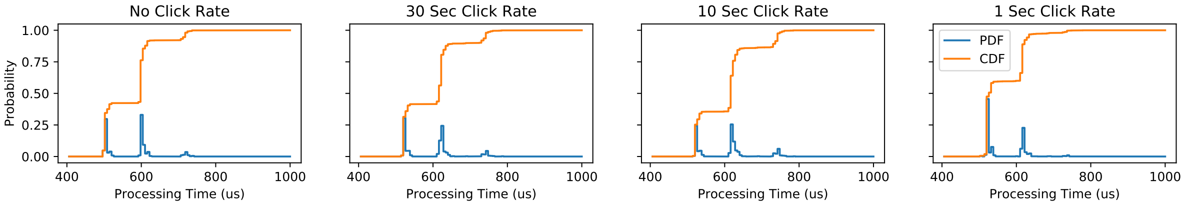


Figure 3.7: The probability distribution (PDF) and cumulative distribution (CDF) of time required for the Real-Time Engine to process audio data packets over one-hour period under different request rates; it is capable to process data with no latency.

Results

Figure 3.7 summarizes the evaluation. In Figure 3.7, we plot the probability distribution (also termed as a probability density function, PDF) of the time it takes the reference design to complete processing one clip of data. RT-MHA receives one clip of data every 1,000 us, which means that the time that the reference design has to complete the processing is bound to a hard upper limit of 1,000 us. Therefore, to assess whether an algorithm is real-time, we need to inspect the worst-case scenario. As we can see from Figure 3.7, the worst-case scenario happens at around 750 us for all four of the runs.

First, this indicates that the reference design can finish computation ahead enough of the deadline. There is also plenty of headroom, which indicates that we can expand the reference design to include more algorithms while still maintaining real-time performance. The second and most relevant information conveyed here is that users' interaction with the system through the EWS has no impact on the real-time processing; therefore, this validates the software design decisions for the RT-Engine in section 3.3.

3.5 Usability Study

In the previous section, we empirically showed that the OSP platform well meets the industry requirements on acoustic features and processing capability. OSP promises a usable hardware basis for hearing health care research. In this section, we shall present a subjective evaluation of the OSP system to determine the acceptability of the OSP device to a person with hearing loss through a structured interview as part of a pilot field study evalutating OSP using the ”Goldilocks” app. This study was designed and ran by our collaborators at the Auditory Research Lab (ARL) at San Diego State University (SDSU) [BKM$^+$19].

The objective of this evaluation was to figure out the usability of the OSP from a hearing aid users point of view in terms of 1) aesthetics, 2) sound quality, 3) ease of use, and 4) whether they are willing to use the device for extended periods of time outside a lab for research purposes. The users were interacting with the OSP system throughout the ”Goldilocks” pilot study, which lasts for two and a half hours. A fraction of the study had the users use the OSP device outside the lab environment, giving the users a diverse experience with the OSP system. At the end of the study, we collected the participants’ feedback and thoughts on the OSP using a structured interview.

In the rest of this section, we will describe the participants and the equipment used in the study. We then outline the procedure that the participants followed. Finally, we present and discuss the results and findings of the study.

3.5.1 Participants

The Institutional Review Board at SDSU approved the study before data collection and written informed consent was obtained from all participants. Twenty one people (twelve male, nine female) were recruited for the study with an average age of 75.1 years (std = 13.9 years). All participants have their own hearing aids, and the group average for how long they have owned a hearing aid is 7.4 years (std = 7.0 years). The group average for the amount of time in any given day the participant uses their hearing aid is 10.9 hrs/day (std = 5.3 hrs/day). All participants were compensated for their time.

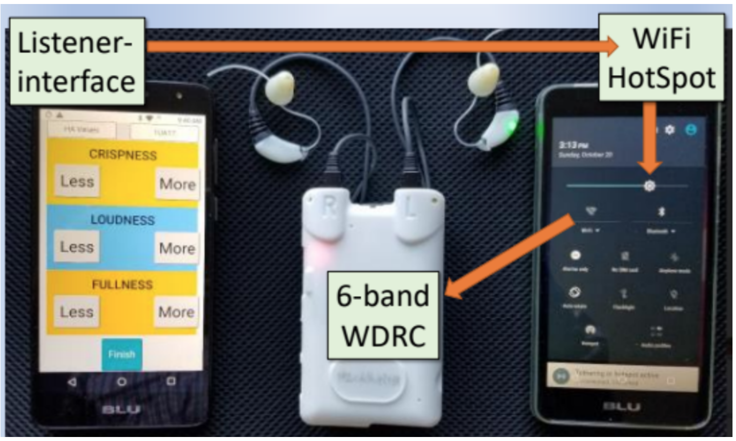


Figure 3.8: Apparatus used for the SFP study. The arrows represent the information flow from the listener interface, via WiFi, to the Portable OSP to tune the WRDC.

3.5.2 Equipment

The experiment was conducted in a room equipped with a sound level meter, recorder, a wifi hotspot, a Verifit device, an OSP portable device with hearing aid, a computer, an amplified speaker, an otoscope, the user's smartphone and some extra batteries. The sound level meter is used to calibrate the amplified speaker. We use the recorder to measure the environmental noise when testing outside of the lab. The portable OSP device is the hearing aid device that the participant will be using. The WiFi hotspot connects the computer, the user smartphone and the portable OSP device together. The Verifit device is used to program the initial starting conditions on the portable OSP device and measure the fitting of the hearing aid. There is a computer for the researcher to program different starting conditions on the portable OSP device and to play stimulus through the speaker. The amplified speakers play the audio stimuli to the participant who will be sitting in front of it during the lab section of the trial. The otoscope is there to check the participant's ears before inserting any object in the canal. The user phone is how the user will interact with the self-fitting app running on the portable OSP system, via the wifi hotspot (Figure 3.8). Finally, there will be extra batteries just in case if one of the battery operated device runs out of power.

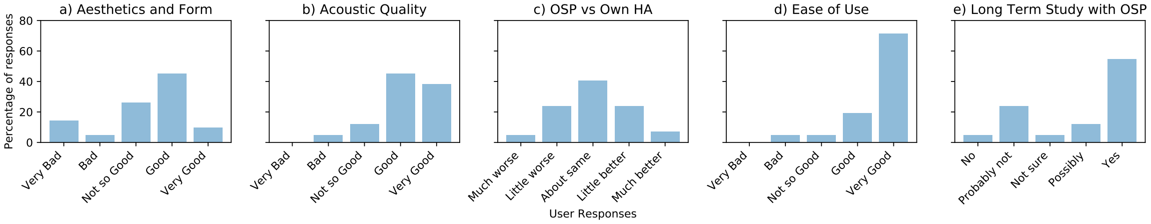


Figure 3.9: Ratings (in histograms) by the participants in the usability study, regarding a) aesthetics and form of the portable OSP device; b) sound quality; c) sound quality of the portable OSP compared to their own hearing aids; d) level of ease-to-use of the web-app; and e) whether they are willing to participate in a research project with the portable OSP.

3.5.3 Procedure

The OSP usability using ”Goldilocks” pilot study occurred in individual sessions over the course of 7 months. Participants were numbered in the order they participated in the study, and experimental conditions were assigned based on odd or even number. First, all participants completed a consent form and intake form. Next, the audiologist measured the participant’s pure tone audiometry (PTA)[3], a metric that quantifies the patient’s hearing loss. The research then programmed two sets of starting condition parameters on the OSP platform for the SFP study. The first was called ”NAL-NL2”, which uses the NAL-NL2 prescription [KDF$^+$11] software to convert PTA into the different parameters used by the hearing aids. The second is called ”GENERIC”, which is the same setting for all participants.

The participant was then handed the phone with the self-fitting app running and was asked to choose either ”NAL-NL2”, if their participant number was odd, or ”GENERIC”, if their participant number was even, for the starting condition. Then the research would play a speech stimuli through a speaker mounted in front of the participant and the participant was asked to adjust their aid using the app. At the end of this period, the new hearing aid fit was saved as ”NALQ1” for odd participants and ”GENQ1” for even participants. The trial was repeated again this time starting with the other starting condition.

After finishing the second trial, the participant was asked to choose ”NAL-NL2” as the

[3] https://www.asha.org/public/hearing/Pure-Tone-Testing/

starting condition. Then the researcher would play a speech plus noise stimuli through the speaker and the participant was asked to adjust their hearing aid. The outcome from this trial was saved as ”NALN1”.

The researcher would ask the participant to select the starting condition ”NALQ1” and would attach an external recorder on the subject. Then they took the participant out of the lab and into a public setting. There the research maintained a conversation with the participant as they adjusted the hearing aid. Once the participant completed this task they saved this fitting as ”OUT1”.

The participant was brought back into the lab and the first three experiments were run again, except this time the results were saved as ”NAL-Q2”, ”GENQ2” and ”NALN2”. This point marked the end of the self fitting portion of the study.

Next, the researcher would evaluate the outcome of the self-fitting by having the participant take a speech recognition test four times once using the ”NALQ2” setting, once using ”NALN2”, once with their own hearing aids and once without any hearing aids. After this, the researcher would measure the amount of correction the hearing aid provides under all of the conditions saved during the trial. This is done by taking the sound quality measurement in the participant’s ear canal using a probe tube while they wear the OSP hearing aids. The researcher records the in ear measurements for all seven saved fitting conditions, as well as, with both the participants own hearing aid and without any hearing aids. This marks the end of the measurement portion of the study.

Lastly, each participant was debriefed using a semi-structured interview. The participants answered questions about the usability of the device in terms of: aesthetics/form, sound quality, long-term usability and ease of understanding the app instruction flow. We next analyze the users’ responses and present our findings.

3.5.4 Interview Results and Findings

About 56% of participants thought that the aesthetics/form of the portable device were either “good” or “very good”. Other participants commented about the size of the device being too big or the device awkwardly swinging when they walk. We learned that 80% of participants thought the acoustic quality of the portable OSP was “good” or “very good”, while 68% of users thought that the sound quality was at least as “good” or even “better” than their normal hearing aids. While 90% of users said that the ease of use of the ”Goldilocks” interface implemented in the EWS was “good” or “very good”. They found the application to be responsive and simple to use both indoors and outdoors. About half of the participants said that they would wear the portable OSP system in a multi-week research experiment. The participants that are willing to do the long-term experiment responded that on average they would be willing to use the device for 3 hours a day for 4 days a week for 2-3 weeks.

We found these results to be encouraging and somewhat expected. Despite receiving commentary from many participants about the portable system being too bulky, over half of the participants said they would possibly or definitely be interested in participating in further research using the portable OSP. We were pleased to learn that most users thought the sound quality and overall experience was good enough that they would be willing to use the system for a research study. This is encouraging because such a study would allow our collaborators to answer very interesting questions in their research. About 90% of participants commented that a big success of this pilot study was the user interface, and a couple of participants noted that the process of self-adjustment was just as easy outdoors as it was indoors. The opportunity to run experiments using OSP in the wild with hearing-impaired users will give researchers a more realistic understanding of their patients’ hearing. Finally, it is clear that our collaborators from ARL was able to adapt OSP to their own needs in this study, and achieved results for their research, that can now be more easily reproduced in the community. ”Goldilocks” app is available as a part of our GitHub repository: https://github.com/nihospr01/OpenSpeechPlatform-UCSD.

3.6 Discussion

We have arrived at a solution which starts to bridge the ”valley of death”. The problem space is still quite vast and daunting. However, by designing this system, we made the problem more tractable. The short development cycles with deliverable kept the project on track and helped obtain plenty of feedback in the early stages. In every cycle we were able to improve and innovate upon the previous system. This has brought us closer to bridging the gap. At the end of our third cycle we were able to evaluate the demo unit both in the lab and in the field.

The results show that the platform we have designed and developed allows algorithm developers to design real-time components without having to worry about the effects of users interacting with the system. At the same time the platform provides UX developers and audiologist a playground to design and develop different interfaces for hearing aid research. This allows the two worlds to co-exist almost in isolation of each other. However, when it comes to integrating the two worlds, the RT-MHA framework provides a simple yet powerful APIs for the task. In our experience with this software model, we have noticed a decrease in development cycle since developers can iterate in isolation from the rest of the system and thus the integration cycle is relatively quick. This needs to be further studied and quantified to understand the impact of this development model.

One of our major concerns while evaluating the portable device using ANSI3.22 test was that the high equivalent input noise would have prevent field studies from occurring until the issue with the codec was fixed. However, we were surprised at how the majority of the participants in the study rated the acoustic quality as good, even when comparing to their own hearing aid. One possible explanation is that the noise is bellow the perceivable range of most of the users. However, participant 1UA08 mentioned that the ”1kHz tone was irritating” but still gave it a ”Good” rating for the acoustical quality. So there must be another factor that needs to be explored.

3.7 Acknowledgements

I want to acknowledge Dr. Arthur Boothroyd's team at San Diego State University: Prof. Carol Mackersie, Paul Grand, Dr. Christine Kirsch, Dannielle Martin, Robert Novak, Shaelyn Painter, and Elena Shur for helping gather the very vital feedback from the users during their pilot field study about the Open Speech Platform. This crucial information helped shape how I designed and developed the Open Speech Platform's software framework.

Chapter 3, in part, is a reprint of the material as it appears in ”Open speech platform: Democratizing hearing aid research.” In Proceedings of the 14th EAI International Conference on Pervasive Computing Technologies for Healthcare, pp. 223-233. 2020. Dhiman Sengupta; Tamara Zubatiy; Sean K. Hamilton; Arthur Boothroyd; Cagri Yalcin; Dezhi Hong; Rajesh Gupta; and Harinath Garudadri. The dissertation/thesis author was the primary investigator and author of this paper.

4 Designing the Real-Time Framework for OSP

4.1 Overview

Many in the open-source audio processing tools for hearing aid (HA) research community feel that the best hardware platform for the next generation of HA research tools should be based on mobile computing platforms[1][2]. These platforms include single board computers (SBC) like the Raspberry Pi, Qualcomm 410c, the Beaglebone, etc. Driven by a intense competition on their use in large number of portable devices, including smart phones, these platforms are constantly evolving to achieve an optimal combination between computation, energy efficiency, programmability and portability. These mobile platforms achieve this by using multiple super-efficient CPU cores, usually two or more, among other processing elements. Therefore, it is essential to design the research tools with the hardware in mind.

This chapter makes four contributions to this area. The first contribution is to analyze the best way to set up the operating environment for these mobile platforms to get the most out of the hardware. We then designed the real-time master hearing aid(RT-MHA) framework using the information gathered from our analysis. RT-MHA is a part of the Open Speech Platform (OSP), designed and optimized to utilize most of the resources available on these platforms. Next,

[1] https://batandcat.com/portable-hearing-laboratory-phl.html

[2] http://www.openmha.org/userproject/2017/12/21/openMHA-on-raspberry-pi.html

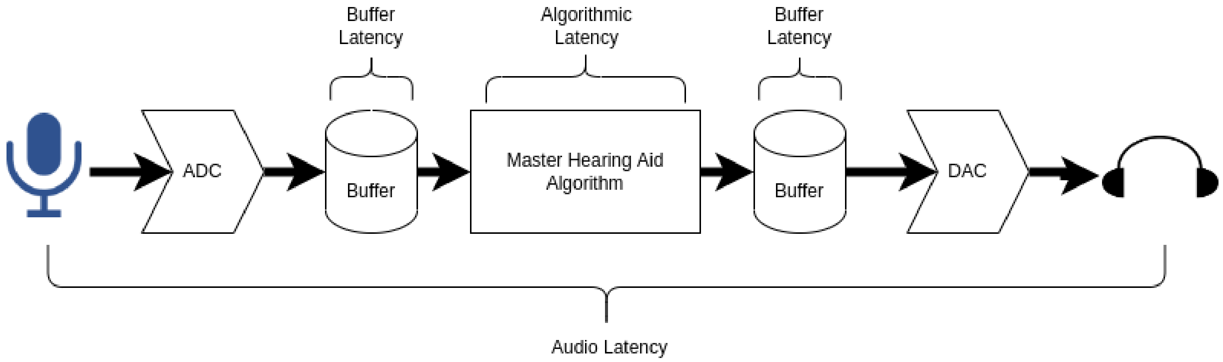


Figure 4.1: The audio latency of a modern day hearing aid consists mainly of the audio buffers latency and the algorithmic latency.

we describe and evaluate the mechanism that allows external applications to interact with the real-time master hearing aid. Finally, we compare the OSP framework against the openMHA framework [KHM$^+$22].

4.2 Background

4.2.1 Defining Real-Time

Given any time bounded task, a real-time system must be able to finish the computation required for that task and respond with the answer within a time period defined by the deadline. For the OSP platform, the main real-time task we are interested in is the hearing aid functionality. To determine the deadline for the hearing aid task, we must define three variables: acceptable audio latency, algorithmic latency, and audio buffer, Figure 4.1.

Audio latency refers to the time it takes audio to be captured by the hearing aid and played back into the listeners' ears, Figure 4.1. This time interval is upper-bounded by the maximum delay that a hearing aid user can tolerate. According to a series of studies the so-called Tolerable Hearing-Aid Delays [SM99, SM02, SM03, SM05, SMMD08] is between 20-30 ms for users with hearing loss. On the other hand, interestingly users without hearing loss can tolerate at most 10ms, especially when it comes to their voice [GCBM18]. Self speech has two routes of travel one through bone conduction and the other through the air. The difference in arrival is what causes the discomfort. People with hearing loss has less input from the bone conduction making it were they are able to tolerate longer latencies. Given these constraints we designed our framework to minimize the audio latency within 10ms.

The timing budget of audio latency consists of two components: one is the algorithmic latency and the other is buffering delay. The algorithmic latency is the time delay between when a stream of audio data enters and exits the signal processing chain, Figure 4.1, a.k.a. group delay in signal processing. The algorithmic latency is unique to the signal processing algorithm. Algorithmic latency defines how many samples the output is delayed from the input, e.g. is the group delay in an FIR filter.

Buffering is crucial to managing overall throughput of processed speech signal. This throughput is maintained at a cost to overall latency that defines the second component. In Figure 4.1, data is streamed from the analog to digital converter (ADC) to a buffer; once that buffer is full, it tells the processor that the following data set is ready to be processed. Then the processor processes that data and needs to finish processing that data before the next set of data arrives, which happens periodically whenever the buffer is full. Similarly, a digital to analog converter (DAC) streams data out of its buffer, and when the buffer is empty, it requests the next set of data from the processor. The size of the two buffers defines the buffer latency, and the amount of time the real-time task has to process the data. For example, if the buffer size is 10 ms, the processor will have 10 ms to churn through the data, and the audio latency will be 20 ms since there is a buffer for the ADC and on for the DAC.

Now we can derive the deadline for the hearing aid task on the OSP real-time system. Let us define the following variables: A_{max} is the maximum audio latency; A is the measured audio latency; α is the algorithmic latency; α_{CPU} is the computation time required for the hearing aid algorithm; β is the audio buffer latency, which also is the task deadline. Using these variables, we can create the following equations.Equation 4.1, describes that audio latency must be less than

Table 4.1: Defines how to interpret the results in the evaluations throughout this chapter.

Metric	**Unit**	**Description** **Key:** L - Lower is Better, H- Higher is Better
Predicted	(μs)	Predicted amount of time required for the design to execute (L)
Min	(μs)	Fastest computation time (L)
Mean	(μs)	Average computation time (L)
Std	(μs)	Variability of the computation time. (L)
Responsiveness	$(Mean - Min)$	How quickly does the system respond to new data (L)
Reliability	$(\frac{Deadline - Mean}{Std})$	How consistent is the system (H)
Missed Deadline	$(\%)$	During testing how many deadlines were missed (L)
Computation Speed	$(\frac{Mean_{Baseline}}{Mean})$	Computation speed measure against the baseline (H)
Power	(W)	Power required by RT-MHA (L)

or equal to the acceptable audio latency of 20-30ms. Equation 4.2, shows the audio latency of a modern-day hearing aid in terms of the audio buffer latency and the algorithmic latency, seen in Figure 4.1. Finally, Equation 4.3 describes that the computation time required for a buffer of data must complete in the time it takes to record the next buffer of data which is our task deadline. Throughout this chapter, we set $\beta = 1ms$ because it is the smallest usable buffer size provided by the PortAudio library for our platform. Also, a smaller buffer allows for bigger algorithmic latency, which more complex signal processing algorithms may require. However, this trade-off will not be tackled in this thesis.

$$A \leq A_{max} \tag{4.1}$$

$$A = \alpha + 2\beta \tag{4.2}$$

$$\alpha_{CPU} \leq \beta \tag{4.3}$$

Before discussing the framework, we need to define the key performance indicators (KPIs) in evaluating our real-time system [All87, Kay16]. Table 4.1 summarizes the different key metrics we use to evaluate the real-time performance of our framework throughout this chapter. Not all evaluations use all metrics defined in Table 4.1. The following defines the three main KPIs we target for our real-time framework for OSP: responsiveness, reliability, and quality of service.

Responsiveness

our goal is to build a framework that is able to assess, verify and guarantee that a given master hearing aid (MHA) implementation meets important timing constraints. Therefore, we define the system's responsiveness as its ability to finish the computation required before the deadline. Which we define as the difference between the average time required for the hearing aid computation and the theoretical fastest time required for the hearing aid computation, see Equation 4.4. By defining the responsiveness, we can evaluate the framework independently of the algorithm's computation need. The system's responsiveness lets us know how much overhead the system and framework have when running the real-time task.

$$Responsiveness = E(\alpha_{CPU}) - Min(\alpha_{CPU}) \tag{4.4}$$

Reliability

A system may be able to hit the deadline of a task most of the time, but a real-time system needs to hit the deadline with a guarantee. We define this real-time guarantee as the reliability of a real-time system. Real-time system reliability falls under three broad categories: hard real-time, firm real-time, and soft real-time. A hard real-time system must meet all of its task deadlines, or else it will result in a system failure. For example, some saws have a mechanism that help prevent bodily injury if that failed to respond in time then it would be catastrophic. Therefore, a hard real-time system needs to guarantee that 100% of the task will meet its deadlines by considering every task's worst-case execution time (WCET).

On the other hand, if a firm real-time system misses its deadline, there will no longer be any utility to that data, but the system can still function normally. Therefore, a firm real-time system can usually be designed using statistical bounds, e.g., 99% of all the tasks on the system must meet their deadlines. The OSP systems hearing aid functionality is a perfect example because if the deadline is missed then an end user will hear artifacts in their audio which is not a catastrophic failure.

Finally, the last category of a real-time system is the soft real-time system, where if a task misses its deadline, there still is a utility for that data. However, the utility decreases over time; otherwise, the system functions as expected. This type of real-time system is what we interact with almost daily; e.g., the interaction with a website can be categorized as a soft-real time task since we expect that the website will respond within a specific time. However, if it takes longer, it just degrades our experience but not the website's functionality.

The OSP system can be defined as a mixed critical real-time system because it requires both a firm real-time guarantee for the hearing aid system and a soft real-time guarantee for the rest of the system (e.g., the embedded webserver). The reason we categorize the hearing aid task as a firm real-time is that if we miss a deadline, the outgoing buffer to the DAC in Figure 4.1 will be empty, causing an anomaly in the audio stream. However, this will not result in a catastrophic failure of the system. This type of failure means we can describe the real-time guarantee for the hearing aid system in statistical terms. Throughout this chapter, we will define KPI for reliability as the ability of the hearing aid task to meet its deadline in terms of how many standard deviations away is the current average from the deadline, the higher, the better. See Equation 4.5 for the equation in how we define reliability.

$$Reliability = \frac{\beta - E(\alpha_{CPU})}{Std(\alpha_{CPU})} \tag{4.5}$$

Quality of Service

The quality of service(QoS) of a real-time system is subjective to the end-user. As mentioned in section 1.2, this platform is targeting three types of researchers: algorithm researchers, hearing aid researchers, and clinical researchers. The algorithm researchers might be looking for the maximum amount of computation they can perform in a given data buffer. Computation in a real-time system can be derived by the amount of time left between when the computation finishes

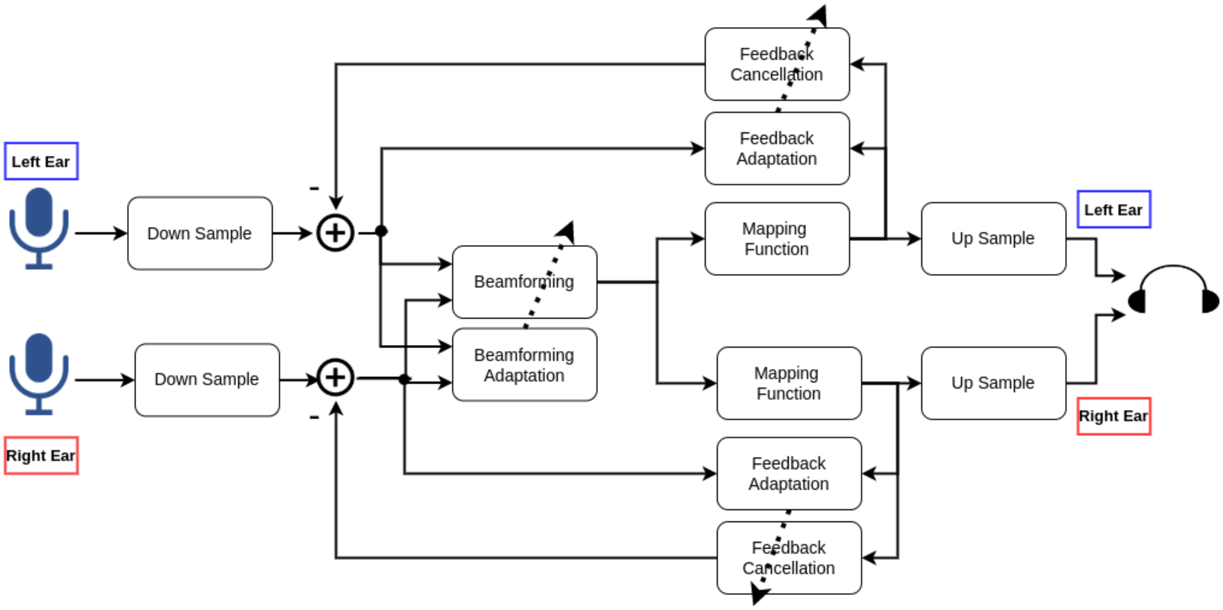


Figure 4.2: Digital signal processing path for the reference RT-MHA implementation, comprised of audio resampling, beamforming with adaptation, sub-band filtering, wide dynamic range compression (WDRC), and feedback path cancellation with feedback adaptation.

and when the deadline occurs. We will define this as real-time responsiveness throughout this chapter. Computation is only one metric for QoS. Hearing aid researchers and clinical researchers may use other metrics like the the system's battery life as their metric of QoS because it impacts the types of experiments they can conduct using these systems. Therefore, depending on the evaluation we will only consider the necessary QoS depending on the trade-offs we are making in each of the sections.

4.2.2 Reference Master Hearing Aid Design

Throughout this chapter we will be using the reference master hearing aid (MHA) design in Figure 4.2 for evaluations. This reference design is based on Kates work [Kat08], plus it incorporates the beamfroming from [LRG17] and adaptive feedback cancelation from [LCZ$^+$19]. The following briefly describes the function of each of the components in the reference design, for more details see [GBL$^+$17, LRG17, LCZ$^+$19] :

- *Down Sample & Up Sample*: These blocks re-sample the audio stream from 48 kHz to an output at 32 kHz and vice-versa.
- *Beamforming*: Beamforming is an advance singal processing algorithm used for reducing noise by preserving sound coming from the front of the user and attenuating sound from other directions.
- *Beamforming Adaptation*: This block periodically estimates the direction of the incoming audio to tune the filters in beaforming algorithm for a better performance.
- *Mapping Function*: This block contains a band-pass filter that separates the incoming signal into one of N different sub-bands for further processing (6 bands in this reference design) and maps the audio in each of the sub-band using Wide Dynamic Range Compression (WDRC) algorithm.
- *Feedback Cancellation*: This block uses the estimated transmission path between the BTE-RIC's output speakers and microphones to reduce ringing from feedback.
- *Feedback Adaptation*: This block periodically estimates the transmission path and updates the filters coefficients in the Feedback Cancellation block.

4.3 Designing the Operating Environment

The performance of a task is heavily dependent on its operating environment. Therefore, carefully designing the operating environment is crucial to best suit the application. The most basic building block that helps us abstract away the details of the hardware and makes it easier to program is an operating system(OS). However, that abstraction usually comes at the cost of real-time determinism.

The lowest level of abstraction is Bare Metal programming, which means everything has to be programmed by hand. There are no abstraction or supporting libraries for the device, making it very difficult to program. However, since the programmer has complete control over the entire hardware stack, it becomes much easier to reason about the real-time performance of the task in a given system.

The next level of abstraction after Bare Metal programming is real-time OSes (RTOSes). The RTOSes are primarily composed of a set of libraries that allow the programmer some abstraction level and provide the bare minimum functionality of an OS, e.g., RTOSes have a real-time scheduler. However, with an RTOS, the programmer must compile everything statically into one binary image that runs on the hardware. Statically compiling everything limits the programmability because the programmer needs to build the environment every time. Even though RTOS has great real-time determinism, the barrier to entry for using RTOS as a research tool for non-computer science experts is too high.

The portable hardware we use, described in section 3.2, has a quad-core ARM CPU. This hardware can use real-time hypervisors like Jailhouse [Ram19], which gets pretty close to the real-time determinism of the RTOS and the programmability of an OS like Linux. Jailhouse is a partitioning hypervisor, which allows multiple OS to run exclusively on different CPU cores and virtualizes the hardware resources among the different OSes running on the system. The virtualization of the hardware is deterministic. Therefore, RTOSes that run on top of Jailhouse can have almost the same level of real-time determinism. However, Jailhouse requires the programmer to create custom communication methods for communications between different OSes. This layer adds complexity to the system making the barrier to entry for a research tool too high again.

Finally, we are left with the tried and tested Linux. For the OSP platform, Linux has the correct abstraction required to develop research tools for researchers working on hearing aids. However, Linux right out of the box is not configured for real-time tasks, meaning that it will not have the best real-time guarantees. The following describes the crucial mechanisms in modern-day Linux used to create a deterministic Linux system for our application.

4.3.1 Configuring Linux for Real-Time

A Linux operating system is not guaranteed to deliver real-time or timely performance. However, several modifications are made to ensure the operating system can meet the timing constraints [ME17]. Even without formal guarantees these changes improve latency and responsiveness of OS actions resulting in many variations of so-called ”real-time Linux” implementations. This section will describe and evaluate the three mechanisms with the most impact on the real-time performance of the RT-MHA framework.

Real-Time Scheduling with Preemption

There are many mechanisms available in Linux through which we can achieve low-latency performance [ME17]. This section will describe and evaluate the three mechanisms with the most impact on the real-time performance of the RT-MHA framework.

The task scheduler is the first mechanism we configure to improve the determinism of Linux. Linux has a few different scheduling algorithms, two of which can meet real-time demands. The round-robin scheduler is one of the scheduling algorithms that meets real-time specifications and is intended for use with a set of periodic tasks. The scheduler loops through the set of tasks one at a time in the same order, making it very deterministic. However, this scheduler does not do well in aperiodic tasks as it wastes CPU time trying to load tasks without active computation requirements—the second type of real-time scheduling algorithm is known as the first-in, first-out(FIFO) algorithm. Like the name suggests, a FIFO scheduler queues tasks when they have work to be done and are executed based on the priority and time they appear, making it deterministic. Our system mixes periodic firm real-time tasks, the hearing aid algorithm, and aperiodic soft real-time tasks, such as the embedded web server. Therefore, the best real-time scheduler for our platform is the FIFO scheduler.

The hearing aid task requires higher priority than the other tasks running on the OSP platform. Therefore, to improve the system’s responsiveness and reliability, the hearing aid task

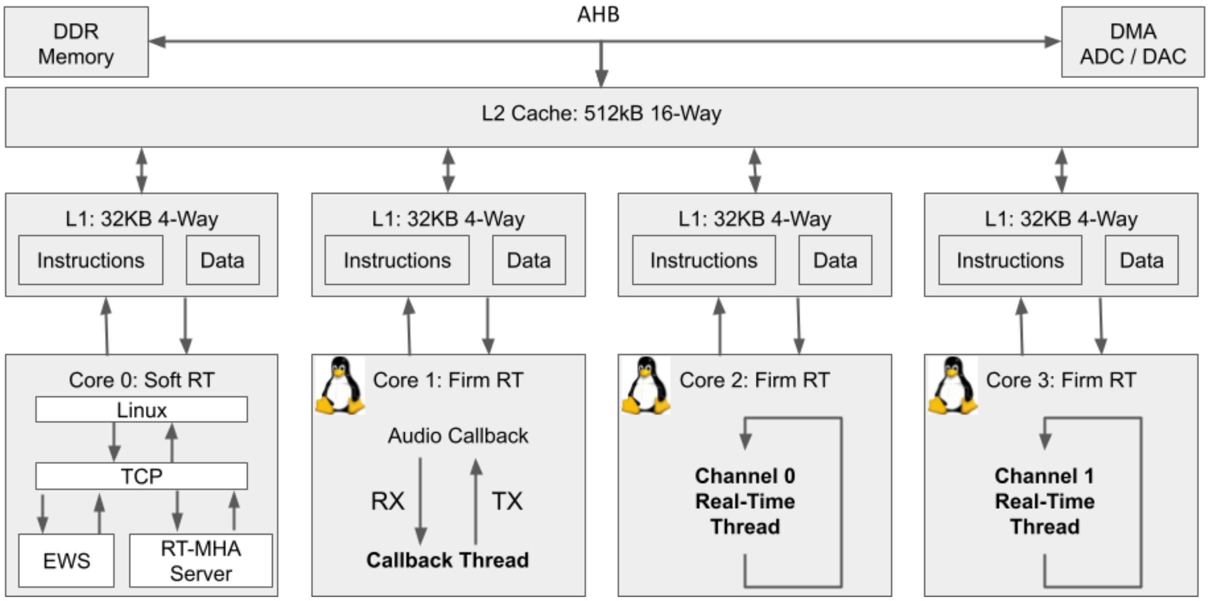


Figure 4.3: This figure shows the partitioning of the CPUs between firm real-time tasks and soft real-time tasks. Plus the figure describes the memory subsystem which is crucial in understanding how data is shared between the different CPUs. Finally, the figure shows the highlevel details of how the OSP software is executed on this hardware.

would need to be able to run as soon as it receives new data from the ADC buffer. In Linux, such a mechanism is a part of the scheduler called preemption. It allows a high-priority real-time task to preempt most other tasks, except for a few kernel-level tasks.

Partitioning the CPU Resources

A Linux system is more deterministic with just a FIFO scheduler and preemption than the stock Linux system. However, that mechanism does not address one of the significant sources of uncertainty. When sharing the same CPU resources between different tasks, the Linux kernel needs to switch the memory context in the hardware based on the currently running task. This context switching ensures memory isolation between different applications running in userspace. However, it comes at the price of real-time performance because context switching means memory has to be fetched into the cache whenever the kernel switches between tasks, creating a significant overhead. In order to solve this issue, we took inspiration from partition-based hypervisors like JailHouse and similarly configured Linux. We partitioned the CPU hardware resources between soft real-time and firm real-time tasks. The firm real-time task will be given three out of the four CPUs, and the soft real-time tasks will be given one of the four, see Figure 4.3. In Linux, a mechanism called IsolCPU allows us to pseudo-partition the CPU hardware resources. IsolCPU completely isolates the CPU resources from the scheduler, and under no circumstances can the scheduler allocate a task on the CPUs listed by the IsolCPU mechanism. The user has to manually pin the tasks they want to run on these isolated CPU resources using another mechanism called CPUAffinity. Therefore, IsolCPU is more deterministic and is the preferred mechanism for partitioning the CPU hardware resources.

Using the IsolCPU mechanism, we have essentially isolated the CPU and memory resources associated with that CPU. Any tasks we run on those isolated CPUs will not be forced to context switch unless the programmer chooses. In order to further improve the determinism of the firm real-time tasks, we will run the firm real-time tasks as a single userspace application. A single userspace application completely prevents context switching on the firm's real-time CPU cores.

CPU Idling

The final mechanism that needs we considered when designing a real-time system is the idling mechanism of the hardware. In modern-day microprocessors, like the Qualcomm 410c, the hardware has a mechanism to turn different parts off to conserve energy. This mechanism is perfect for power efficiency. However, it wreaks havoc on firm real-time tasks. One way to mitigate this issue is by running a background task whenever the firm real-time task is finished computing, which will prevent the micro-processor from idling. However, this would mean that the microprocessor will burn more power than it requires. Therefore, in this chapter, we will propose a way of distributing the workload to help mitigate some of the uncertainty caused by the idling mechanism.

4.3.2 Evaluation

In the rest of this chapter, we will be running the evaluation on the updated version of the portable hardware mentioned in section 3.2. The OSP hardware is a custom solution built around the Qualcomm system on a module (SoM), which consists of a quad-core ARM A53 chipset with 1 GB of RAM and 8 GB of memory. Figure 4.3 best describes the CPU resources available on the OSP hardware platform that matter to the firm real-time framework. For more detailed information about the hardware, please read [PWZ$^+$19].

We split the evaluation of the three mechanisms into two parts. The first part evaluates the scheduling and isolCPU mechanisms since these two mechanisms require a soft real-time load on the CPU. On the other hand, the CPU idling mechanism only requires the running of the firm real-time task, which we evaluate in the second part.

Scheduling and IsolCPU Mechanism

Setup: In this evaluation, we ran three different configurations of Linux, starting with the stock configuration. Next, we enabled the real-time FIFO scheduler with preemption. Finally, we isolated three of the four CPUs to dedicate them to the firm real-time task. We tested all of the configurations using two processes. The first process was the firm real-time task running the algorithm described in Figure 4.2 on a single CPU. Using only one of the CPUs for the firm real-time task simplifies the experiment and ensures that any changes in the results are mainly due to the mechanism. The second process we ran was a stress test which acts as our soft real-time load. The stress test simultaneously creates an artificial load on the CPU, memory, and hard drive. This test represents the worst-case soft real-time application and its impact on the firm real-time task. Each task ran for a total time of 10 minutes per configuration, and we recorded the time it takes for the firm real-time task to process a buffer of data from when it was released. The data buffer size is 1ms of audio, which means that the deadline to finish processing the audio is also 1ms.

Result: Table 4.2 best summarizes the results of this experiment. Figure 4.4 represents that information as a box plot showing the distribution of the processing time it took the firm real-time task to compute a buffer of audio data. In terms of all three KPI criteria, the stock did the worst, then came the FIFO real-time scheduler with preemption, and the best performant was the IsolCPU with the FIFO real-time scheduler plus preemption. These results show the benefits of using these mechanisms on the firm real-time tasks. However, these benefits come at the trade-off of the percentage of available resources for the soft real-time task. With the IsolCPU mechanism and how we set up our operating environment, soft real-time tasks will always have 25% of the resources. While with the FIFO scheduler with preemption, soft real-time tasks will receive between 5% to 100% of the resources, which get dynamically allocated by Linux depending on the firm real-time workload.

Another thing to note in these results is that the Min metric stayed relatively consistent no matter the configuration. This consistency is because, given the hardware and the algorithm, the Min represents the absolute fastest the hardware can run the algorithm.

CPU Idling

Setup: To understand the impact of the CPU idling, we created a simple background that runs a while loop to prevent the CPU from idling. This way, we can isolate the impact of the CPU idling on the firm real-time tasks. In each set of experiments, we will compare the real-time performance of only the firm real-time task running, allowing the CPU to idle, and the firm real-time task with the background task, preventing the CPU from idling. Each task will run for a total time of 10 minutes per configuration, and we will record the times it takes for the firm real-time task to process a buffer of data.

For this evaluation, we will run two sets of experiments to understand the impact of the CPU idling on firm real-time tasks with different computation loads. The first set of experiments will compare the impact of the CPU idling on a hearing aid algorithm with just the mapping function by disabling the adaptive feedback cancellation and the beamforming algorithms in the signal processing algorithm in Figure 4.2. The second set of experiments will compare the impact of the CPU idling on a hearing aid algorithm designed in Figure 4.2.

Result: The results from this experiment is summarized in Table 4.3 and visually displayed in Figure 4.5 as box plots. As we can see from the results, the idling mechanism has a big impact on the real-time performance of the real-time task, no matter the CPU load. However, according to our experimentation, more CPU load equates to a better response and reliability when allowing the CPU to idle. This result makes sense as the entire CPU has less time to idle. Therefore, the system has a better real-time performance. These results tell us that the best way to know the system's capabilities is to test it with an idle function. Then depending on the requirement of the end-user, the idling function can be enabled or disabled.

4.3.3 Discussion

In this section, we have evaluated the effects of the different mechanisms on the real-time performance of the system. We expected that the system would achieve better responsiveness and reliability as we dedicated more resources and removed the interruptions. However, what surprised us was the idling mechanism's impact on real-time performance. We can see that real-time performance improves proportionally with the CPU load by testing the idling mechanism under different CPU loads. This improvement is contributed to the increased CPU load preventing the CPU from idling for a more extended period. Therefore, idling time seems to be inversely proportional to real-time performance. In the next section evaluation, we consider the power penalty of the idling function as a design consideration.

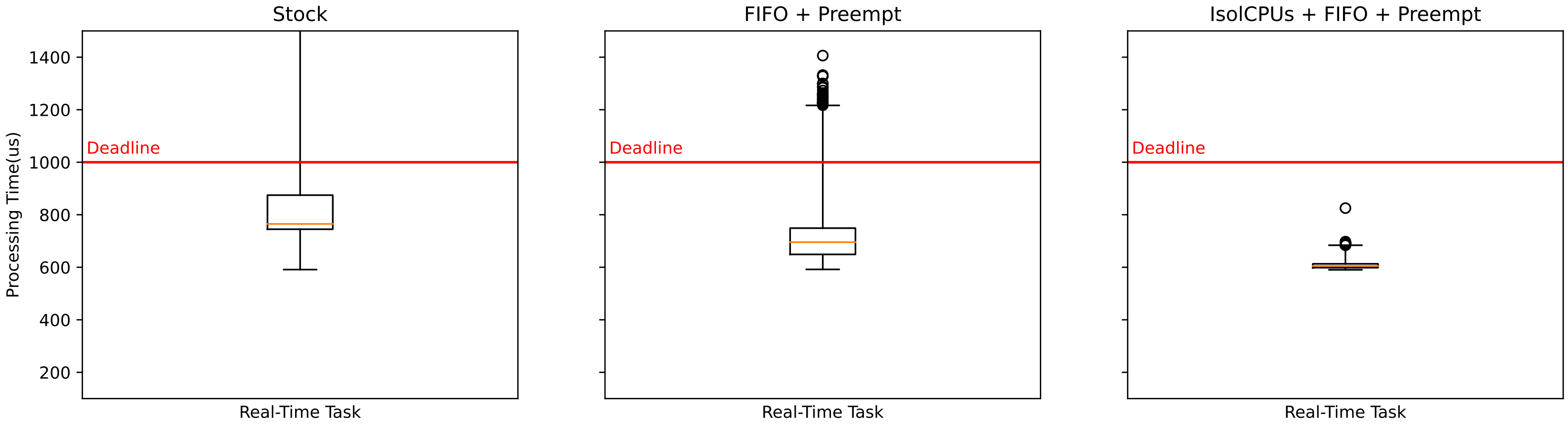


Figure 4.4: These set of graphs show the impact of the different Linux Mechanisms on Real-Time.

Table 4.2: This table summarizes the impact the FIFO real-time scheduler with preemption and the CPU isolation mechanism has on the real-time performance of the OSP framework. The highlighted numbers in red are the best performance in this set of values.

Metrics	**Units**	**Stock**	**FIFO with Preempt**	**IsolCPU & FIFO with Preempt**
Min	(μs)	591.25	591.88	**590.39**
Max	(μs)	3864.5	1648.9	**825.26**
Mean	(μs)	890.82	709.00	**608.02**
Std	(μs)	294.33	81.28	**12.67**
Responsiveness	$(Mean - Min)$	299.57	117.12	**17.63**
Reliability	$(\frac{Deadline - Mean}{Std})$	0.371	3.58	**30.93**
Missed Deadlines	$(\%)$	15.74	0.938	**0**

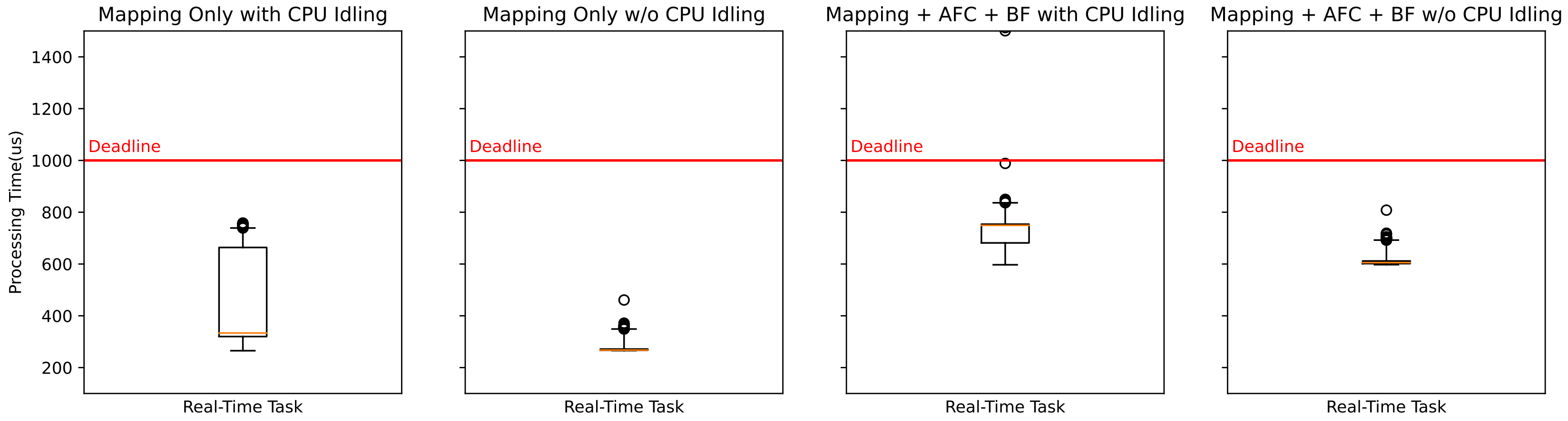


Figure 4.5: These set of graphs show the impact of letting the CPU go into idle given different workloads.

Table 4.3: This table summarizes the impact the CPU idling mechanism has on the real-time performance of the OSP framework given two different work loads. The highlighted numbers in red are the best performance in each set of values organized by their respective workload.

Metric	Units	Mapping Only		Mapping & AFC & BF	
		With CPU Idling	Without CPU Idling	With CPU Idling	Without CPU Idling
Min	(μs)	**265.21**	265.63	**597.08**	597.56
Max	(μs)	758.23	**460.99**	1525.1	**808.04**
Mean	(μs)	431.00	**271.76**	721.64	**611.41**
Std	(μs)	169.15	**9.87**	57.19	**15.72**
Responsiveness	$(Mean - Min)$	165.79	**6.13**	124.56	**13.85**
Reliability	$(\frac{Deadline-Mean}{Std})$	3.36	**73.78**	4.87	**24.72**
Missed Deadline	(%)	**0**	**0**	0.001	**0**

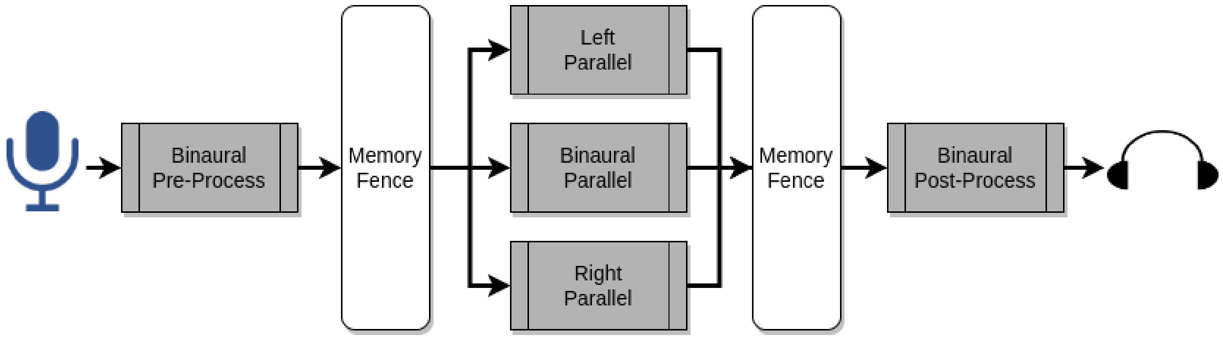


Figure 4.6: Proposed domain based partitioning scheme created using the ontology of hearing aid algorithms.

4.4 Work Load Distribution

Now that we have set up the operating environment for the OSP platform, the next step is to properly distribute the firm real-time task across the three firm real-time CPU cores, Figure 4.3. The firm real-time task of the OSP platform, a.k.a. the real-time master hearing aid algorithm (RT-MHA), is an ensemble of modular digital signal processing library modules. An example of an RT-MHA algorithm is Figure 4.2, section 4.1 describes each component of this RT-MHA algorithm. We will use this example throughout this section to measure the real-time performance of the different partition schemes in the OSP framework.

4.4.1 Manual Partitioning using Domain Knowledge

Here we will consider an intuitive and programmer-friendly approach to partitioning the hearing aid algorithm amongst the three CPU resources. A hearing aid's algorithm can be split into three groups depending on the data stream required. The first group is the algorithms that require data streams from both the ears and from here on out, we will call it the binaural group. The other two groups only use one of the ears data stream, which we will label as either the left or right group depending on the data stream it is processing.

Analyzing a set of hearing aid algorithms [Kat05, KVdBMW07, DGM$^+$08, KLHL09, Kat19, SSC$^+$21, SSH$^+$22], we designed the partition scheme described in Figure 4.6. This

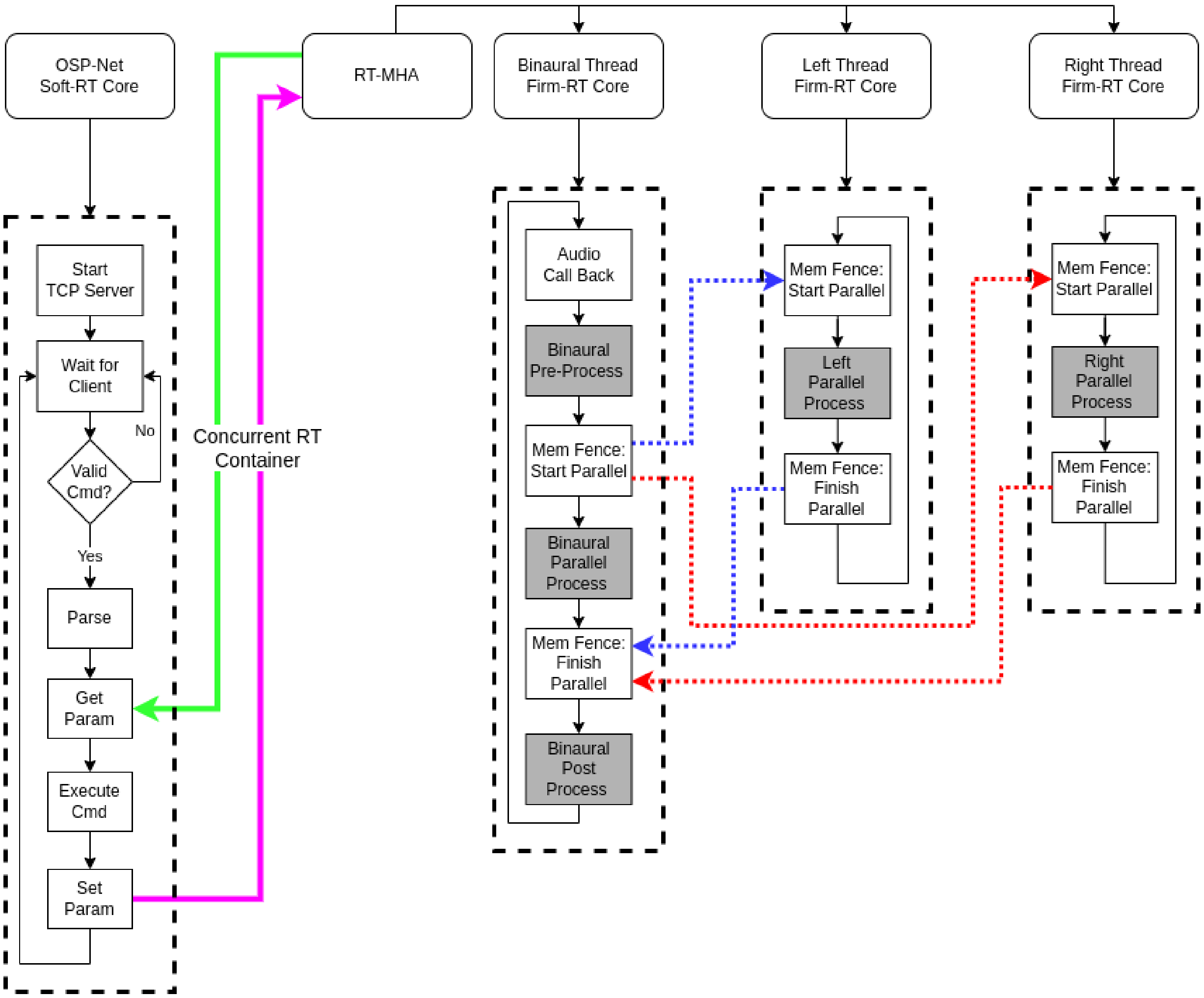


Figure 4.7: Describes how the domain based partitioning scheme runs on the hardware in Figure 4.3

partition scheme splits the computation into three distinct sections. First, microphone data from both ears goes to the pre-processing section, where most binaural processing is done on the data streams. The following section is the parallel processing. Here we process the left and right channels independently, plus we process any adaptation algorithms required by the binaural processing for the next cycle. Lastly, we have the post-processing section, where data from both the channels are combined and sent to the ADC.

This structure allows us to create an abstract class upon which programmers can design their hearing aid algorithm. The essential advantage of programming using this abstract class is that the framework will take care of all the memory fencing required, Figure 4.7. Plus, the abstraction allows hot-swapping between different algorithms, enabling researchers to compare and contrast many hearing aid algorithms. Another advantage of designing the data flow over the other partitioning schemes is the ability to run the same data flow sequentially because there are no data dependencies between the three parallel processing parts in the middle, making this framework device agnostic well. In the next chapter, we discuss this point in further detail.

Figure 4.8 is the result of partitioning the example hearing aid process defined in Figure 4.2 using the domain-based partition scheme. In this example, CPU 0 is the binaural thread, CPU 1 is the left thread, and CPU 2 is the right thread. As one can see from the figure, there is symmetry between CPU 1 and CPU 2. This symmetry is one of the key reasons why it is easier to program and think about partitioning the algorithm using this scheme. According to this schedule, we should expect $\sim 306\mu s$ end-to-end timing. We do not expect this partitioning scheme to be the most computationally efficient. It will depend heavily on the hearing aid algorithm. The main purpose of this partitioning scheme is for better programmability compared to more optimized partitioning schemes. Parallel programming is not easy, so to reduce the programmer's burden, we created a partitioning scheme with the least number of memory fences required while getting the most out of the multiple CPU resources.

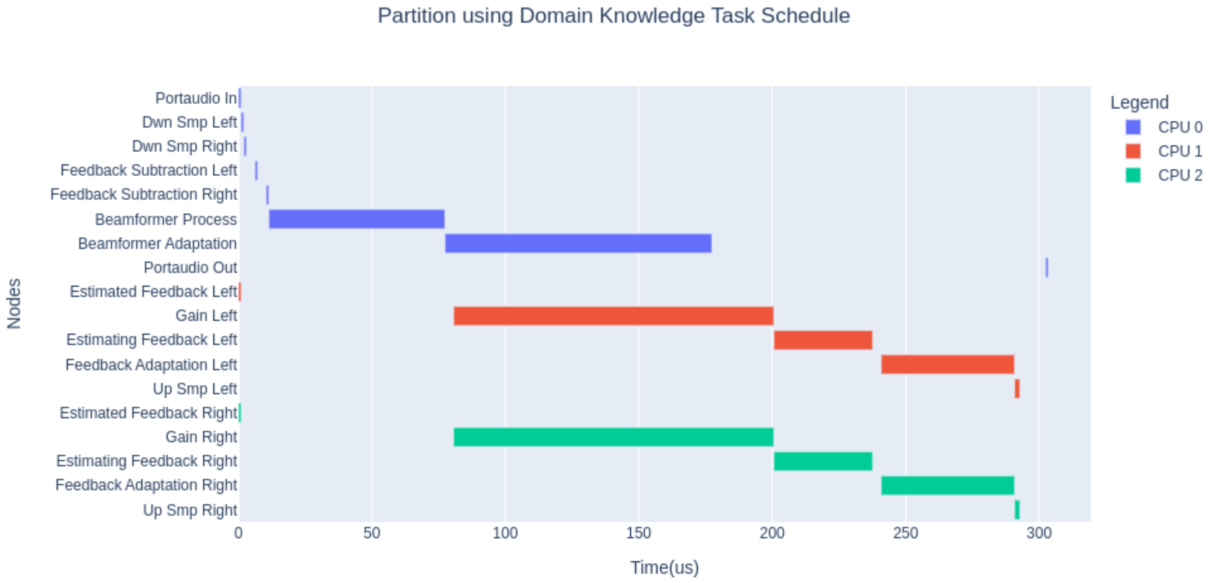


Figure 4.8: This figure describes how the signal processing algorithm Figure 4.2 should be run on the hardware according to the programmability first partitioning scheme.

4.4.2 Automated Partition of RT-MHA using Directed Acyclic Graph

We can automate the partitioning process because the signal processing chain in Figure 4.2 has a one-to-one mapping to a dependency graph that can be used for workload distribution. The graph can be partitioned into three segments denoting the different CPU resources. However, we first need to ensure that the dependency graph we create from the signal processing algorithm is acyclic. In Figure 4.2, we can see that the Feedback Cancellation creates a cyclic dependency. We can break the cyclic nature of this graph by only considering one period of the signal processing chain. By only looking at one period, we can split cyclic functions into two components: the first is the component that consumes the data from this period, and the second is the producer that produces the data consumed in the previous period. This way of splitting the cycles means we can now take the signal processing chain and map it to a directed acyclic graph (DAG), which we will be partitioning into three segments. Figure 4.9, shows the DAG of the signal processing algorithm described in Figure 4.2. The dashed line in Figure 4.9 is a weak link to show that the

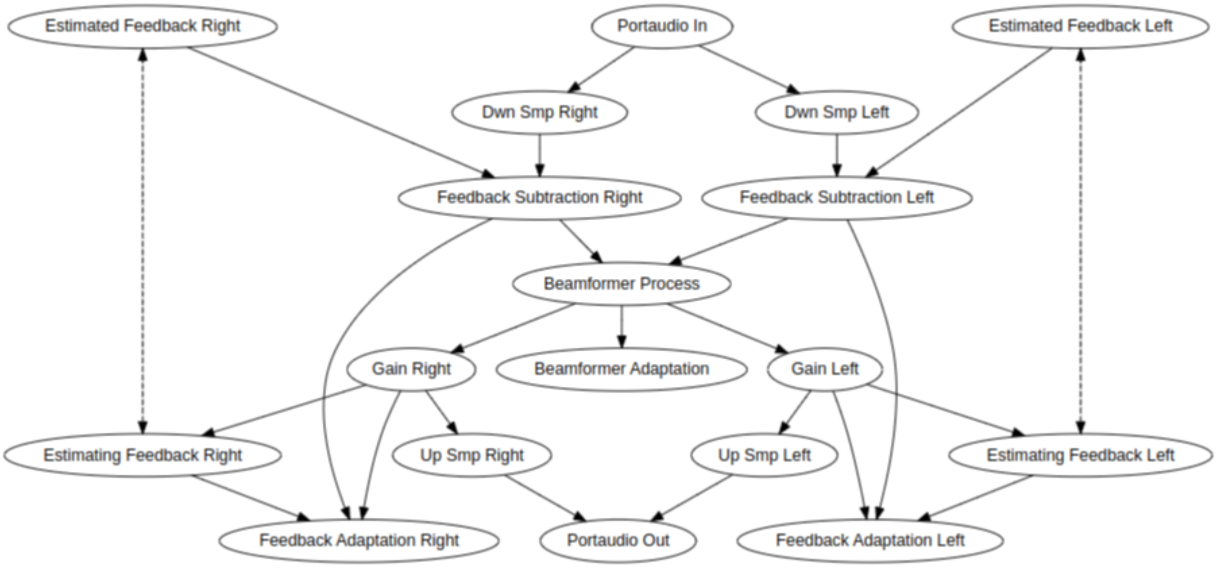


Figure 4.9: DAG representing the RT-MHA algorithm as a dependency graph to be used for partitioning into the different CPU resources.

two nodes are part of the same Feedback Cancellation algorithm.

In order to best segment the DAG, we first need to define the information in the graph. The graph consists of nodes and edges. Each node is a producer, a consumer, or both. The function of each node determines the information the node requires for the partitioning algorithm. The information is as follows:

Processing Time - Defines the total amount of time in μs required to process the data that the node consumes. If the node is a producer and not a consumer then this time will be set to $1\mu s$.

Memory Transfer Time - Defines the total amount of time in μs it takes to move data produced by a node from one CPU level-1 cache to another CPU level-1 cache. If the node is a consumer and not a producer, this time is set to $1\mu s$. This penalty will only be applied once, and we assume the memory will be available on both the CPU level-1 cache.

Group Id - This field allows us to group the nodes to force the partitioning algorithm to assign all the nodes in the same group to the same CPU. If this field is left blank, we allow the partitioning algorithm to choose any CPU on which to run this node.

Processing CPU[3] - This is an informational field used by the partitioning algorithm. When the process has been assigned, a CPU to run on it is recorded here. After the entire algorithm is run, we can use this field to describe the algorithmic flow of the hearing aid algorithm on the OSP hardware platform.

Memory Location[1] - This field is used to determine whether the memory transfer penalty needs to be applied or not. It keeps track of where the data produced by this node is available for use by the nodes that get executed afterward.

Overall Timing[1] - This field keeps track of the node's starting and finishing time of the processing done. It is used throughout the partition process to understand when the nodes dependent on the current node can start their process.

Now that we have defined the DAG completely, we will define some restrictions and assumptions we make during the partitioning process. They are as follows:

1. One of the three CPU partitions must start and end with the audio callback. Any processing after the audio callback sends data to the DAC must be performed on the other two CPU partitions.

2. A node is runnable if and only if all of the nodes it depends on have finished.

3. We assume that all the data required for the hearing aid algorithm can fit in the level-1 cache.

4. We assume that the penalty for moving data from one CPU level-1 cache to another CPU level-1 cache will only apply once.

The rest of this section discusses the two algorithmic ways in which we can partition the hearing aid algorithm.

[3] Populated after the partition algorithm is completed.

QoS Optimized Partition

The first partitioning algorithm we will discuss optimizes the system's QoS. This partitioning algorithm, Algorithm 2 uses a priority queue to schedule the next runnable tasks. The algorithm chooses the highest priority runnable task and runs it on a CPU that offers the earliest starting time. The node's priority is determined by taking the sum of the processing time for all the nodes reachable by the current node. After we run the highest priority node on the queue, the algorithm updates the priority queue with all the new runnable nodes. It continues doing this until there are no more nodes to run. This algorithm efficiently schedules the nodes across the 3 CPU cores to get the most computation out of the resources. Figure 4.10 is an example of how this algorithm would schedule the reference design Figure 4.2 across the hardware resources. According to the algorithm, we should expect around 303μs end-to-end timing from this partitioning scheme, which is more efficient than the manual partition method. However, the QoS method requires eight memory fences scattered throughout the algorithm, making it very hard to run sequentially. Thereby, this partitioning can only operate using three threads of execution.

Algorithm 2 QoS Optimized Partitioning Algorithm

```
$N$ is a set of all Nodes
$E$ is a set of all Directed Edges
$R$ is a priority queue of all runnable Nodes
while $R \neq$ Empty do
    $r \leftarrow$ top of $R$                                    ▷ Highest priority runnable node
    Run $r$ on the CPU with the earliest start time              ▷ Including memory transfer penalty
    for all $e \in E$ where $e = r \rightarrow n$ and $n \in N$ do
        if $n$ is runnable then
            Priority $p_n = \sum_{\upsilon \in N}$ Processing Time of $\upsilon$, where $\upsilon$ is reachable from $n$
            $R \leftarrow n$ with Priority $p_n$
        end if
    end for
end while
```

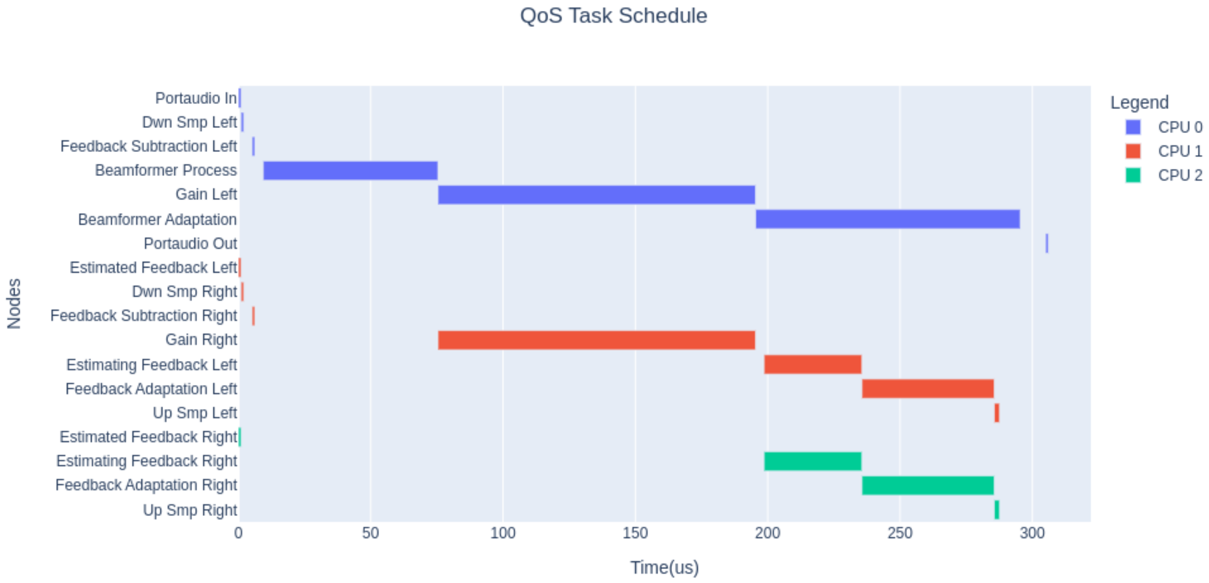


Figure 4.10: This figure describes how the signal processing algorithm Figure 4.2 should be run on the hardware according to the QoS optimized partition algorithm in Algorithm 2

Critical Paths Optimization

Suppose the system's reliability is the limiting factor, e.g., the CPU idling causes uncertainty. We can employ the second partitioning algorithm, which optimizes the critical path of the algorithm before partitioning the rest of the nodes. The critical path in our DAG comprises simple paths that go from the audio input to the audio output and are partitioned first. Other nodes outside these simple paths, e.g. all of the adaptations in Figure 4.9, are partitioned afterward. In Algorithm 3 we create two priority queues, one for critical and another for non-critical nodes. Then the algorithm partitions the nodes exactly like the QoS algorithm, except it prioritizes the critical path nodes. By scheduling the nodes in this fashion, we can hide the uncertainty because we can send data to the DAC before all the hearing aid algorithm processing is finished. The downside to this partitioning methodology is that it is not as efficient as the previous method. Mainly due to the restriction that one of the CPU resources will no longer be available after the data is sent out to the audio output. Figure 4.11 is an example of how this algorithm would schedule across the hardware resources. According to the critical path algorithm, we should expect around 354μs end-to-end timing from this partitioning scheme, which is 17% worse than the QoS optimized algorithm. However, the critical path timing is around 203μs, which is 33% faster than the QoS optimized algorithm. Therefore, this partitioning should be able to absorb some of the uncertainty, especially in a system optimized for power efficiency.

Algorithm 3 Critical Path Optimized Partitioning Algorithm

```
$N$ is a set of all Nodes
$C = \{c : c \in N$ where c is a critical node$\}$
$E$ is a set of all Directed Edges
$R_C$ is a priority queue of all runnable Critical Nodes
$R_{NC}$ is a priority queue of all runnable Non-Critical Nodes
while $R_C \neq$ Empty & $R_{NC} \neq$ Empty do
    if $R_C \neq$ Empty then
        $r \leftarrow$ top of $R_C$                        ▷ Highest priority runnable critical node
    else
        $r \leftarrow$ top of $R_{NC}$                     ▷ Highest priority runnable non-critical node
    end if
    Run $r$ on the CPU with the earliest start time    ▷ Including memory transfer penalty
    for all $e \in E$ where $e = r \rightarrow n$ and $n \in N$ do
        if $n$ is runnable then
            Priority $p_n = \sum_{\upsilon \in N}$ Processing Time of $\upsilon$, where $\upsilon$ is reachable from $n$
            if $n \in C$ then
                $R_C \leftarrow n$ with Priority $p_n$
            else
                $R_{NC} \leftarrow n$ with Priority $p_n$
            end if
        end if
    end for
end while
```

4.4.3 Evaluation

This section discussed three types of workload distribution schemes and evaluated them using a simulation-based on the profiling times of the individual library components. Here we evaluated the different real-world schemes on the OSP platform [PWZ$^+$19].

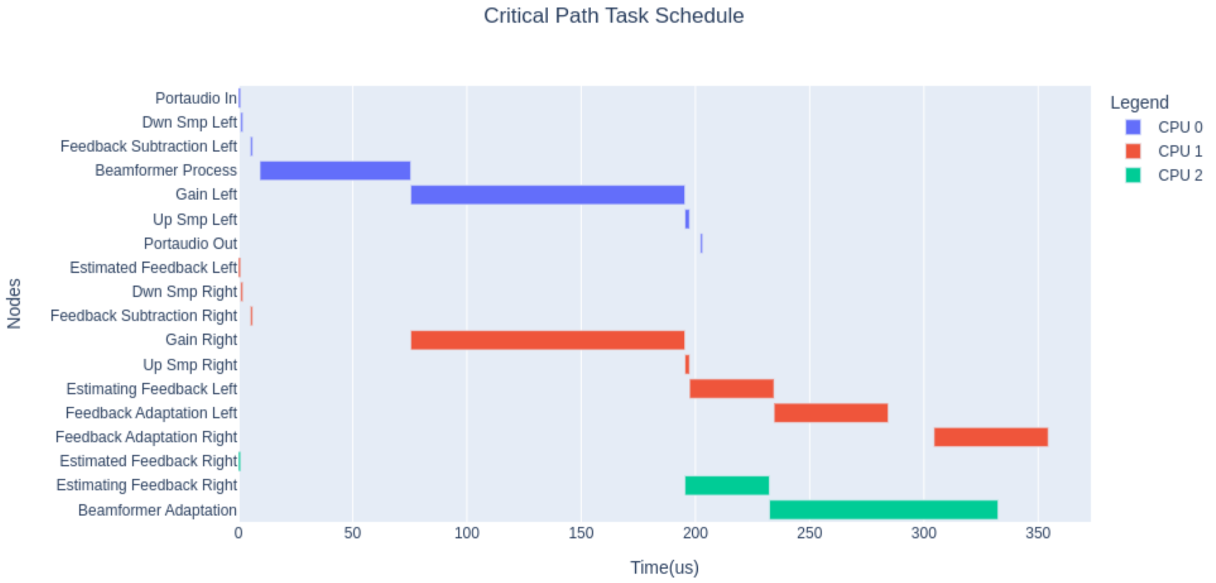


Figure 4.11: This figure describes how the signal processing algorithm Figure 4.2 should be run on the hardware according to the Critical Path optimized partition algorithm in Algorithm 3

Setup: In this evaluation, we compare the three different partition schemes discussed in this section plus the single thread baseline version. We tested all configurations with and without a background task to prevent the CPU from idling. The algorithm described in Figure 4.2 is the hearing aid algorithm we used in all configurations. Each task will run for a total time of 10 minutes per configuration. We recorded the times it takes for the firm real-time task to process a buffer of data and the average power consumed by the device for each configuration.

Results: The results of this evaluation is summarized in Table 4.4 and visually displayed in Figure 4.12 & Figure 4.13 as box plots. The following are the key insights we gathered from the results:

- The Min metric is the theoretical fastest the hardware can process the data given the algorithm and partitioning scheme. The Min metric confirms this because we observe that the value does not change between CPU idling and not idling.

- The predicted run-times of the different partitioning methods were within $10\mu s$ when compared to the fastest computation time measure (Min).

- When we prevent the CPU from idling, the QoS partitioning method does the best in terms of overall real-time performance. The only metric that it is not the best at is power.

- The baseline had the best real-time performance in terms of responsiveness, reliability, and power when the CPU was allowed to idle. This result reiterates that shorter idle time equals more real-time certainty, as we discovered when evaluating the idling mechanism.

- When we prevent the CPU from idling, the power consumed by the RT-MHA framework is inversely proportional to the computational time required. This relationship is because the task preventing the CPU from idling must run only when nothing else is running. Therefore, it has to run for less time when the methods require longer computation time. The results show that the baseline does the best in terms of power when we disable the idling.

- Since all of the methods do the same amount of computation, we observe that the power numbers are nearly identical. The only reason for the difference is that the automated partitioning methods have more memory transfers between CPU cores. However, when the CPU can idle, the partitioning scheme has minimal impact on the power consumption.

- The intuition behind the Critical Path Partition method did not pan out according to Table 4.4. However, after looking at the results in Figure 4.12 we noticed that compared to the other partitioning methods, the critical path looks like it is acting as a buffer. However, from the long whiskers, it seems like the aggressive idling can get triggered right after the critical path is finished processing. Therefore, requiring the non-critical computation to be completed on the next cycle.

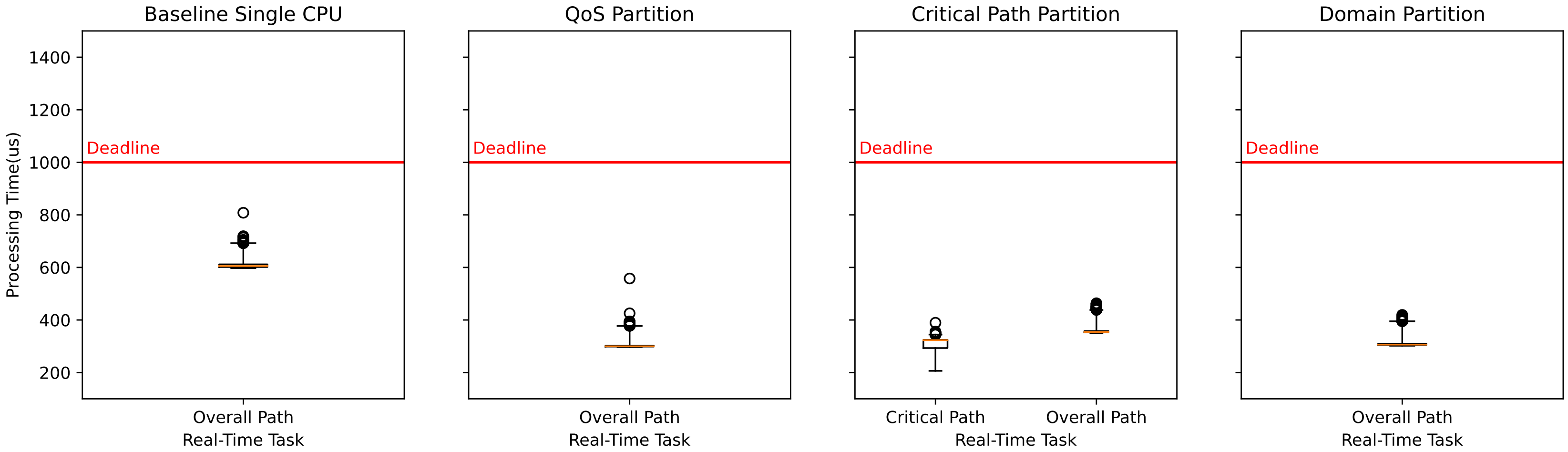


Figure 4.12: Shows the real-time performance with CPU idling **disabled** of the different partitioning methods plus the baseline.

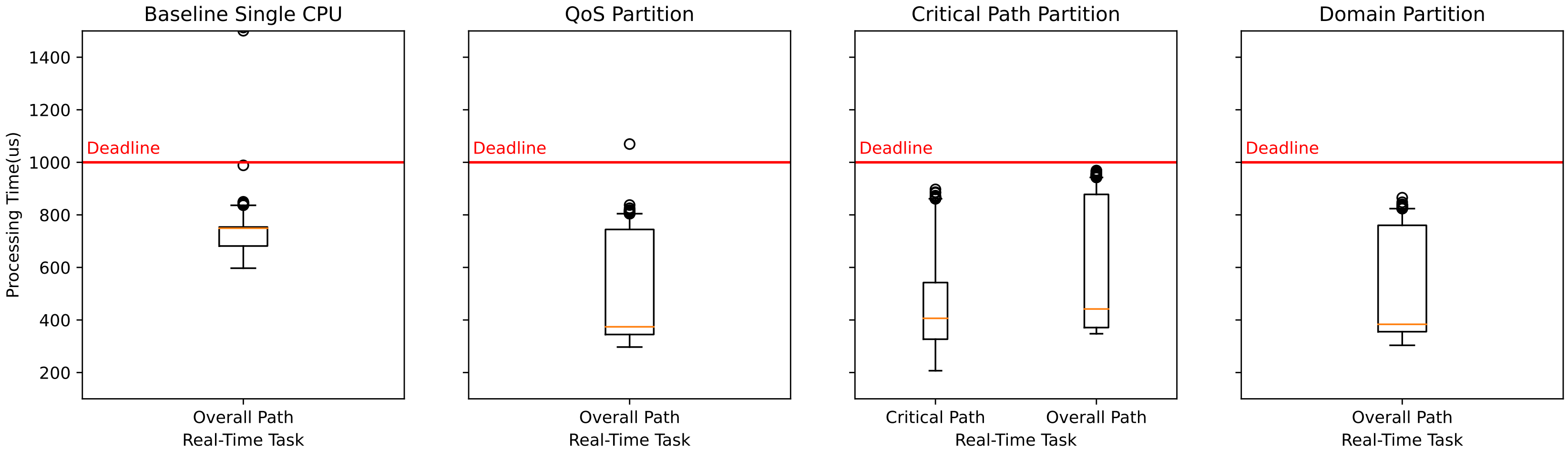


Figure 4.13: Shows the real-time performance with CPU idling **enabled** of the different partitioning methods plus the baseline.

Table 4.4: The outcomes of the evaluation for the different partitioning methods on the OSP hardware. All of the partitioning methods were tested twice once with idling disabled and once with idling enabled. The bolded numbers represent the best result achieved for the metric given the setup. The red highlight numbers are there to differentiate the critical path, since that is only used to determine the reliability of the Critical Path Partitioning method.

		Without CPU Idling					**With CPU Idling**				
				Critical Path Partition					**Critical Path Partition**		
Metric	**Units**	**Baseline Single CPU**	**QoS Partition**	Critical Path	Overall Path	**Domain Partition**	**Baseline Single CPU**	**QoS Partition**	Critical Path	Overall Path	**Domain Partition**
Predicted	(μs)	592.00	**303.00**	*203.00*	354.00	306.00	592	**303**	*203*	354	306
Min	(μs)	597.56	**296.82**	*206.35*	349.17	301.98	597.08	**296.98**	*206.93*	347.66	303.59
Mean	(μs)	611.41	**303.10**	*296.25*	357.62	309.89	721.64	**483.29**	*474.65*	567.19	511.06
Std	(μs)	15.72	**8.84**	*48.69*	9.53	9.73	**57.19**	190.36	*206.94*	227.00	197.05
Responsiveness	($Mean - Min$)	13.85	**6.28**	*89.90*	8.45	7.91	**124.56**	186.31	*267.72*	219.53	207.47
Reliability	($\frac{Deadline-Mean}{Std}$)	24.72	**78.82**	*14.45*	67.40	70.90	**4.87**	2.71	*2.54*	1.91	2.48
Missed Deadline	(%)	**0.00**	**0.00**	*0.00*	**0.00**	**0.00**	0.0013	0.003	*0*	**0**	**0**
Computation Speed	($\frac{Mean}{Mean_{Baseline}}$)	1.00	**2.02**		1.71	1.97	1	**1.493**		1.27	1.41
Power	(W)	**0.52**	0.68		0.64	0.68	**0.42**	0.44		0.44	**0.42**

4.4.4 Discussion

This section presented a few ways of partitioning the workload amongst the available hardware resource, one manual method and two automated methods using directed acyclic graphs. We can draw a few conclusions from our analysis of these partitioning methods. First, domain-based partitioning is the best for manual programmability, and in certain situations, it is almost as good as QoS partitioning. This partitioning scheme makes the most sense when designing an MHA by hand unless the MHA design is too big for this scheme. It is the best for most cases because the framework automatically handles the fencing required for transferring data between the different cores. Plus, it makes the MHA design device-agnostic as this partitioning scheme's running can be sequential.

Next, the QoS partitioning will fit the MHA design most efficiently if all processes are completed in one pass. Therefore, an MHA design that is not able to fit using this method will not be able to fit in the given hardware as is.

The baseline single-core implementation is the best in terms of power for all evaluations and real-time performance when the CPU is allowed to idle. We can extract from this that counter-intuitively, due to the idling mechanisms in this mobile computing hardware, the best partitioning algorithm is the one that fits the MHA design while being the most inefficient in terms of utilizing the computation time available.

Finally, from running the evaluation for the critical path partitioning, it seems that once the critical path finishes, the CPU becomes aggressive with the idle mechanism and sometimes prevents the non-critical modules from finishing. This aggressive idling is the reason for the long whiskers of the critical path partitioning in Figure 4.13. Therefore, we can gather from the results that the best partitioning method is the one that is the least efficient at packing tasks within the real-time computation resource.

4.5 Interacting Between Soft-RT and Firm-RT

So far, the focus has been on getting the most out of the resources for the firm real-time tasks. However, one of the most important aspects of a hearing aid research platform is the ability to freely tune and tweak the many parameters within the real-time hearing aid algorithms without worrying about their impact on real-time performance. In order to enable this functionality, we need to design a mechanism to interact between the soft real-time and firm real-time parts of this framework, e.g., the interaction between the embedded web server and the RT-MHA.

We only care about reliability KPI and its impact on the QoS since the interactions are intermittent for the interaction between soft real-time tasks and firm real-time tasks. The most prominent nondeterministic element of a Linux environment, outside of idling CPU hardware, is memory allocation and de-allocation. Each application in a Linux environment gets its virtual memory that maps to physical memory. When the application tries to allocate memory, it first checks if the virtual memory has a mapping to physical memory. If the memory is not mapped, then the application asks the Linux kernel to do the mapping, and then after that is done, it can use that space. Similarly, when memory is done being used by the application, the kernel may want to reclaim the mapping to free up the memory for other applications. Therefore, making memory allocation and de-allocation nondeterministic depends on the Linux kernel.

For this reason, we strongly advise against allocating and de-allocating memory in the process section at the library module level, Algorithm 4. All allocation and de-allocation of memory should be done in the constructor or the set parameters functionality. We can use the set parameters functionality for these operations because it runs in parallel as part of a soft real-time thread running on the soft real-time CPU resources, as shown in Figure 4.6. Therefore, having minimal impact on the firm real-time task if the internal data transfer mechanism is designed correctly.

Algorithm 4 Modular library skeleton for signal processing algorithms.

```
class libModule{
public:
    ...
    /*@brief Setting libModule parameters*/
    void set_param(...){
        /* Load the current parameters*/
        libModule_param_t const* currParam = globalParam.get();
        /* Create a new data structure and update its values*/
        libModule_param_t* nextParam = new libModule_param_t();
        ...
        /*Update the global parameters*/
        globalParam.set(nextParam);
    }
    /*@brief Getting libModule parameters*/
    void get_param(...){
        /* Load the current parameters*/
        libModule_param_t const* currParam = globalParam.get();
        /* Return all of the parameters by reference*/
    }
    /*@brief Real-time processing inside libModule*/
    void process(...){
        /* Load the current parameters*/
        libModule_param_t const* currParam = globalParam.get();
        /* This is where the real-time code will go*/
    }
private:
    struct libModule_param_t {
        /* Set of Parameters */
    };
    /*The pointer to the global param structure*/
    concurrent_rt<libModule_param_t> globalParam;
};
```

4.5.1 Concurrent Real-Time Data Container

There are two significant ways to deal with concurrency when the set parameters function changes the parameters, and the RT-MHA function uses it. The first and most common way is to use a Mutex lock to safeguard critical sections of code that deal with the shared data. However, the problem with this method as it can cause the soft real-time task to prevent the firm real-time task from executing because if the soft real-time task obtains the lock first and takes a non-deterministic amount of time to finish, it will cause issues for the firm real-time task. Therefore, a better alternative is to use an atomically changed data structure to have the least impact on the firm real-time task.

Current best practices in the audio industry are to use atomic shared pointers to share data between soft real-time tasks and firm real-time tasks [Dou15]. This coding practice assumes that the data contained in the shared pointer is, in essence, immutable because then the programmer assumes that the data contained within the pointer will not change during the real-time processing. Anytime the parameters need to be updated, the programmer needs to allocate a new data structure and atomically update the shared pointer once the data is ready.

In C++, the shared pointer is de-allocated when the references to the pointer go to zero. This mechanism can impact the firm real-time section of the code if it invokes the deallocation when the last one to release the shared pointer is the firm real-time task. In order to prevent this, the programmer needs to create an Adhoc garbage collector, which will always be the last to release the shared pointer. This way of programming requires the programmer to remember to call the garbage collector in the soft real-time section of the code periodically, or else there will be a memory leak.

This way of coding requires much discipline from the programmer. However, for a research platform, we cannot assume that the programmer will adhere to these practices. Therefore, to make programming in a multi-thread, multi-core real-time system more intuitive, we designed a concurrent real-time container called concurrent_rt. This container handles all the intricacies we mentioned while providing a simple get and set functionality for the programmer, Algorithm 5. The *get* function in this container is firm real-time safe. The *set* function, on the other hand, should only be called in the soft real-time sections. The *get* function is real-time safe and can be invoked anywhere.

Algorithm 5 Concurrent Real-Time Data Container

```
template<typename T>
class concurrent_rt {
    ...
    /*@brief Puts the new data into the shared pointer as
     an immutable object*/
    void set(T const new_data) {
        std::shared_ptr<T const> nextParam(new_data);
        std::atomic_store(&globalParam, nextParam);
        /*Updates and runs the garbage collector*/
        releasePool.add(nextParam);
        releasePool.release();
    }
    /*@brief Gets the immutable object from the shared pointer*/
    std::shared_ptr<T const> get() {
        return std::atomic_load(&globalParam);
    }
private:
    std::shared_ptr<T const> globalParam;
    GarbageCollector releasePool;
};
```

4.5.2 Evaluation

Here we evaluate the impact of the interaction between the soft real-time and firm real-time tasks in the real world on the OSP platform [PWZ+19].

Setup: In this experiment, we ran the hearing aid algorithm described in Figure 4.2 on just a single CPU to isolate the impact of the interaction between the soft real-time process and the firm real-time process. The OSP device was connected to an external computer over a WiFi connection. The external computer would send a request to *get* and *set* the parameters on RT-MHA through the RT-MHA Server as quickly as possible. We ran the experiment twice, once with a background

Table 4.5: Shows the results from the evaluation of the interaction between the Soft Real-Time part of the system and the Firm Real-Time part of the system. he bolded numbers represent the best result achieved for the metric given the setup.

		Without CPU Idling		**With CPU Idling**	
Metrics	**Units**	**Baseline without Soft Real-Time Interaction**	**Baseline with Soft Real-Time Interaction**	**Baseline without Soft Real-Time Interaction**	**Baseline with Soft Real-Time Interaction**
Min	(μs)	590.4	**589.7**	**590.1**	591.2
Max	(μs)	825.3	**800.9**	1525	**970.9**
Mean	(μs)	608.0	**599.3**	**721.6**	747.4
Std	(μs)	12.67	**8.61**	57.19	**26.93**
Responsiveness	$(Mean - Min)$	17.66	**9.60**	**117.6**	156.13
Reliability	$(\frac{Deadline-Mean}{Std})$	30.95	**46.54**	4.87	**9.380**
Missed Deadlines	$(\%)$	**0.0**	**0.0**	0.0013	**0.0**

task preventing CPU idling and once without. Each of the experiments ran for a total time of 10 minutes per configuration, and we recorded the times it takes for the firm real-time task to process a data buffer.

Results: Based on the results we gathered from this evaluation, see Table 4.5, we can conclude that there is negligible impact on the real-time performance due to the interaction between soft real-time and firm real-time tasks. Moreover, there seems to be an improvement especially in terms of the reliability metric. This improvement is most probably due to two factors. First, due to the soft real-time application continuously refreshing and updating the parameter space for RT-MHA, it is less likely that the parameters will be evicted from the cache making it more deterministic. Second, the interaction increases the amount of computation done on the device, which usually results in less idle time. Less idle time means better determinism, which is what we see in the reliability metric when we enable CPU idling.

4.6 Framework Comparison

Figure 4.7 best describes all of the different components of our RT-MHA framework and how they interact with each other. In this section we will compare the RT-MHA framework developed for the OSP platform with the openMHA framework [KHM$^+$22]. The openMHA framework was developed with a different set of design constraints. Even though the openMHA can run on single board computer (SBC) platforms like the OSP hardware, the framework was not optimized to take advantage of the full resources available on such SBC. The openMHA was primarily designed so that hearing aid algorithms can be configured at run-time, while being device agnostic. This means that their framework uses dynamic libraries and is restricted to a single thread execution. On the other hand, the OSP framework was designed with SBC architecture in mind, while some trade-offs were made to make it more programmable and device agnostic. The OSP framework requires algorithms to be statically compiled before run-time but is able to take advantage of the multi-core architecture. Given these differences in the design considerations we wanted to evaluate mainly the overhead of each framework on a single thread application.

4.6.1 Evaluation

Since each framework has completely different set of library components, we will only be evaluating the frameworks by themselves. Using a simple gain function, as the hearing aid algorithm, we evaluate the overhead incurred by each of the frameworks on the Qualcomm 410c platform.

Setup: In this experiment we setup both the frameworks with a simple gain function so that we can measure the overhead of the framework by itself. Each of the frameworks were run for a total of 10 minutes. We recorded the system performance statistics. To ensure that CPU idling is not a factor in our evaluation we ran a background task on a CPU that is not being utilized.

Result: We can see from the results in Table 4.6 that openMHA has significantly more overhead than OSP. This difference is mainly due to hearing aid designs being statistically defined at compile time in OSP, while in openMHA it is defined dynamically at run time. By statistically defining the hearing aid designs we are able to create a light weight framework and focus on improving real-time performance of the hearing aid platform. However, this design choice is less

Table 4.6: Shows the results of profiling the two different master hearing aid frameworks running a similar workload. The results show that openMHA has significant overhead compared to OSP.

Metrics	openMHA	OSP	$\frac{\text{openMHA}}{\text{OSP}}$
CPU Utilization (%)	17.55	5.18	3.38
Context Switches per Second	1,974.75	491.76	4.02
Page Faults per Second	5.97	3.08	1.94
Branches (M/s)	6.26	2.85	2.20
L1 Data Cache Loads (M/s)	16.81	6.40	2.63
L1 Instruction Cache Loads (M/s)	39.23	11.85	3.31
Memory Usage (%)	1.0	0.7	1.43

scalable. We address this downside in the next chapter of this dissertation. All in all this result does indicate that our design choice will allow researchers to do more on resource constraint mobile computing platform.

5 Extending the OSP Framework

5.1 Overview

In the previous chapter, we discussed the cores of the real-time portion of the OSP framework. After developing, testing, and deploying different master hearing aid (MHA) algorithms using the OSP framework, we developed the following MHA design flow Figure 5.1. This design flow starts with a specification of a hearing aid design described by the researcher, usually in the form of a signal processing flow. That signal processing flow is then converted to a directed acyclic graph(DAG) using the method we discussed in the previous chapter. Then, using either the manual domain-based partitioning scheme or the automated QoS partitioning scheme, we partition the workload over the available hardware resources, as discussed in the previous chapter. Next, using the signal processing flow and the segmentation we derived from the DAG, we generate the MHA design in C++ for the RT-MHA framework. This design is then loaded into the RT-MHA framework and deployed on the embedded system. Finally, we would test and debug the design in the actual device.

Our investigations discovered the need for the OSP platform to run multiple MHA designs on the device at run-time and for a way to debug the designs after deploying them on the device. In this chapter, we describe how we extended the framework to be more scalable and the framework's functionality to debug the designs in-situ.

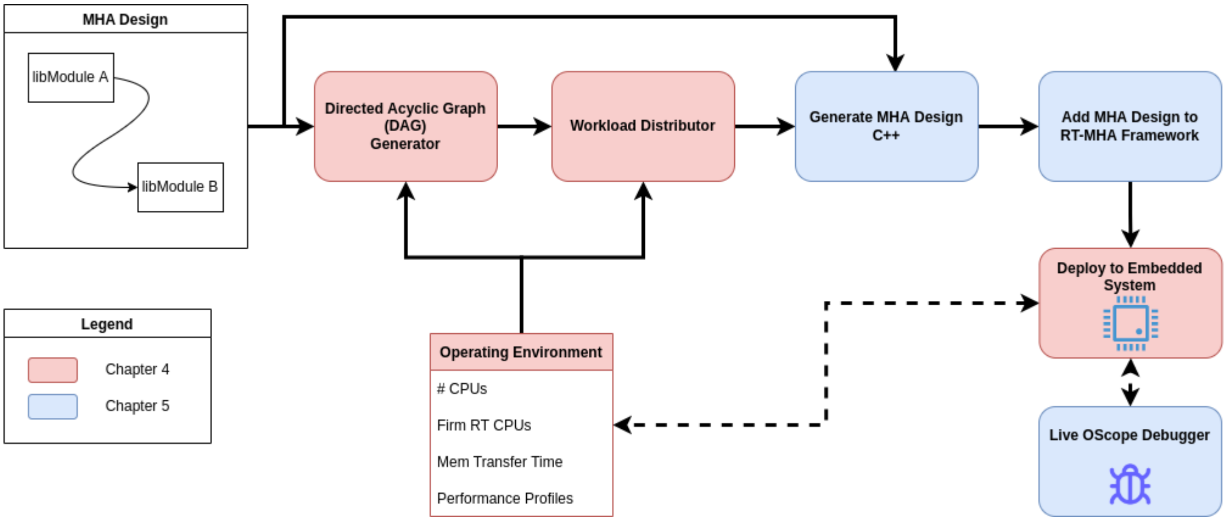


Figure 5.1: Master Hearing Aid design development flow for the OSP framework.

5.2 Scalability

Algorithm 6 MHA Design Skeleton

```
class MHA_Design: public mha {
public:
    /* Constructor and Destructor*/
    ...
    /*Binaural Pre-Process*/
    void binural_pre_process(float *const*dataIn_, size_t buf_size_);
    /*Binaural Parallel Process*/
    void binural_paral_process();
    /*Channel Parallel Process
     /*   + channel = 0 is Left
     /*   + channel = 1 is Right*/
    void channel_paral_process(size_t channel);
    /*Binaural Post Process*/
    void binural_post_process(float **dataOut_, size_t buf_size_);
    ...
};
```

Using the domain-based partitioning scheme we used in the previous chapter as our basis, we create the skeleton of the MHA designs. Algorithm 6 describes the abstract class and is represented by the gray boxes in Figure 5.2, which represents the RT-MHA framework computation flow. This abstraction allows the framework to interpret which MHA design to use at

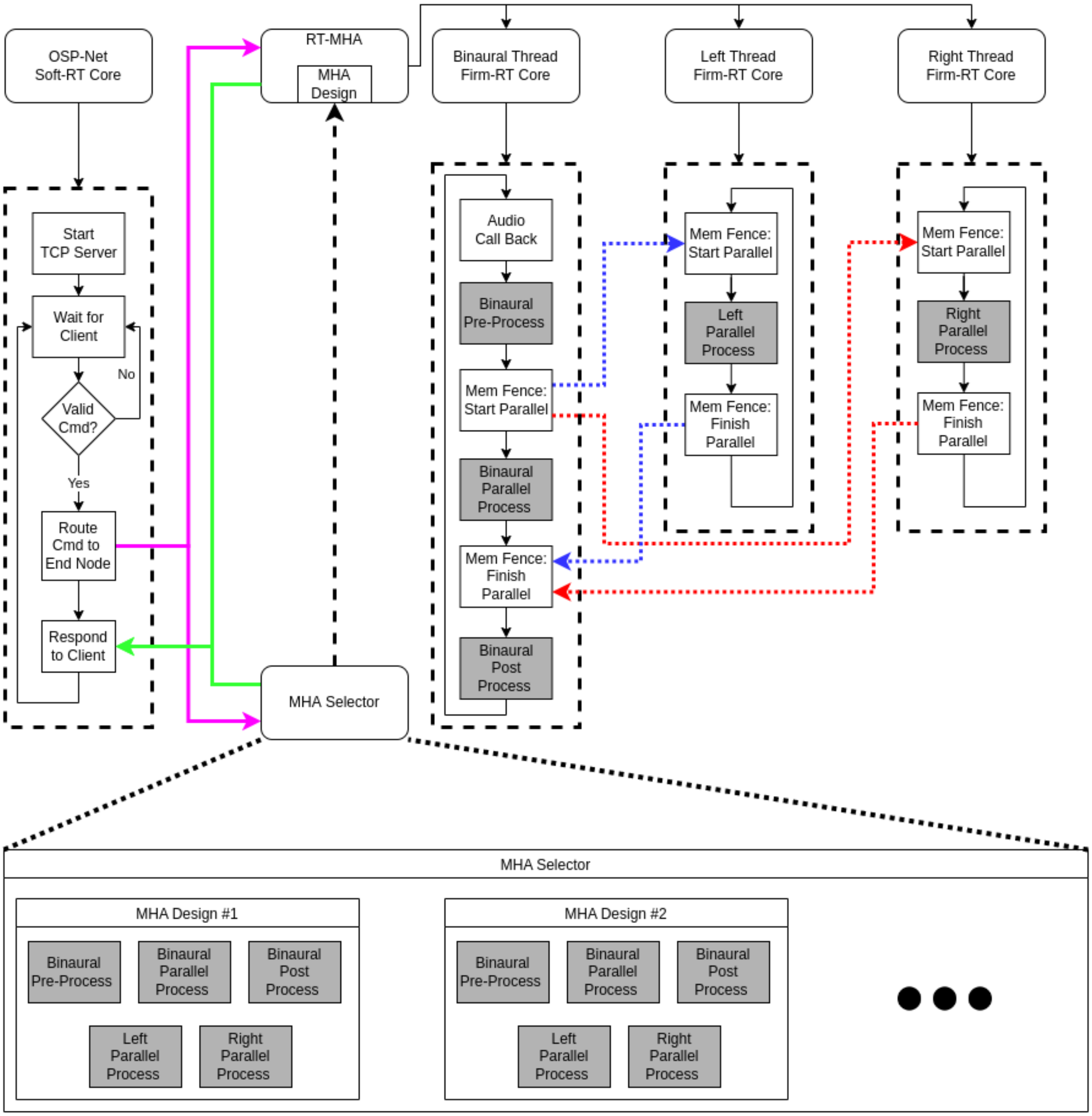


Figure 5.2: Describes the RT-MHA Framework with the inclusions of the MHA Selector and a scalable OSP-Net used as the control plane for RT-MHA.

run-time through polymorphism. To take advantage of this ability of the MHA designs, we create a mechanism that can load different MHA designs at run-time called MHA Selector. Figure 5.2 also describes how the MHA Selector integrates with the rest of the framework. Algorithm 7 is an example of how an MHA design gets loaded and unloaded using the MHA selector.

To fully enable this feature, we need to create a scalable and standardized method for creating APIs so that the external applications can interact with the various MHA designs. Another challenge with reconfiguring the MHA algorithm at run-time, a.k.a. hot-swapping, is the framework's ability to safely change between algorithms because it requires both allocating and deallocating MHA algorithms, which causes significant issues to the real-time operations. In the rest of the section, we will first describe the system that allows MHA designers to create an application programming interface(API) called Rapid API. Next, we describe the hot-swapping mechanism to safeguard real-time operations. Finally, we will evaluate this mechanism and discuss the results.

5.2.1 Rapid API

One of the essential utilities of this framework is its ability to interact with external applications. These interactions allow the users of the device to tune the functionality of the hearing aid process. Therefore, it is crucial to define an API for each of the hearing aid algorithms. We developed a system called Rapid API that allows MHA developers to rapidly create an API for their hearing aid algorithm. The Rapid API system is designed based on the ontology of a hearing aid algorithm and focuses on scalability. The data structure we choose to utilize for representing the data inside the API is the JSON data structure [Fri19]. JSON is a human-readable version of a dictionary data structure, making it descriptive. Plus, it is language-independent and can be structured hierarchically, all things we wanted in the Rapid API. Also, the JSON library from Niels Lohmann [Loh22] made JSON a first-class object making it very easy for researchers to develop the parser required to get the parameter information. In order to make the API scalable,

Algorithm 7 MHA Selector

```
mha* set_mha(size_t selection, mha *current_mha) {
    if(current_mha != nullptr){ // Delete Current MHA
        // Save Parameters for persistence
        current_mha->get_param(osp::mha_config[osp::current_mha] );
        delete current_mha;
        current_mha = nullptr;
    }
    // Load New MHA
    osp::current_mha = selection;
    switch (selection) {
        case 1:{
            std::cout << "6 Band Multicore Implementation" << std::endl;
            if(osp::mha_config[selection].is_null()){
                // Load Default Parameters Here Can Be Left Blank
            }
            return new six_band_multi_thread(...); //MHA Designs
        }
        case 2:{
            std::cout << "Gain Only" << std::endl;
            if(osp::mha_config[selection].is_null()){
                osp::mha_config[selection]["left"]["mic_gain"]["gaindB"]  = 10;
                osp::mha_config[selection]["right"]["mic_gain"]["gaindB"]  = 10;
            }
            return new gain_only(...); //MHA Design
        }
        ...
        default:{
            osp::current_mha = 0;
            std::cout << "6 Band Single" << std::endl;
            if(osp::mha_config[selection].is_null()){
                // Load Default Here Can Be Left Blank
            }
            return  new six_band_single_thread(...); //MHA Design
        }
    }
}
size_t get_mha(){
    return osp::current_mha;
}
void get_mha_info(json &mha_info){
    mha_info[0] = "6 Band Single Core Implementation";
    mha_info[1] =  "6 Band Multicore Implementation";
    mha_info[2] = "Gain Only";
    ...
}
```

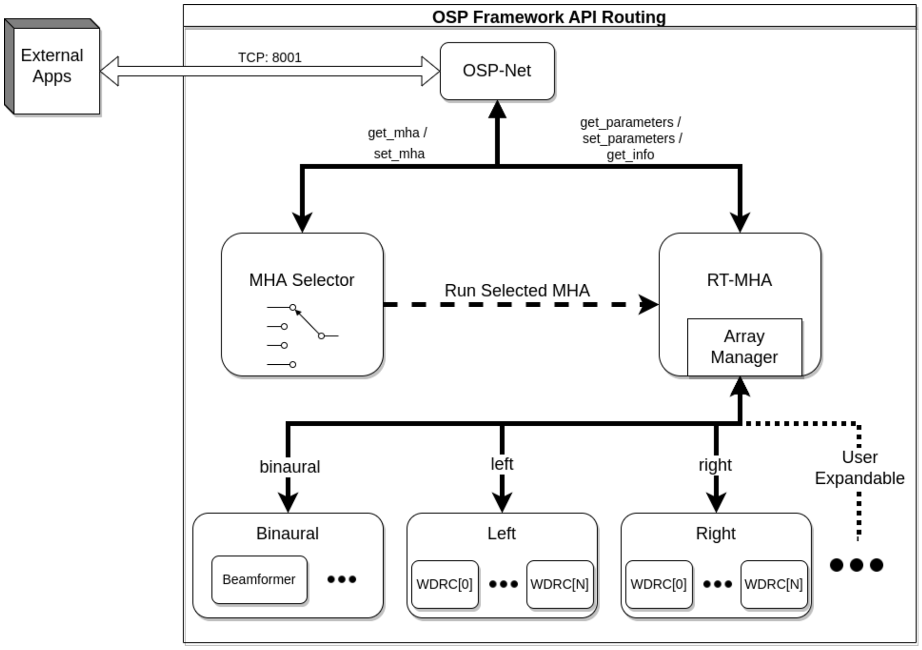


Figure 5.3: Shows how a JSON message is routed through the OSP Framework.

we took advantage of the hierarchical nature of the JSON data structure and designed the routing of the JSON data based on the ontology. Figure 5.3, describes how the data gets routed from external applications to the different end nodes in the OSP framework. The following describes each of the modules in more detail with different examples.

OSP-Net

All communication to the OSP-Net, containing a TCP server, is initiated by the external apps. The external app sends a JSON data structure containing two data fields to OSP-Net. First is the ”method” field describing the action the application wants the RT-MHA framework to perform. The other field is an optional ”data” field to send relevant information to the RT-MHA framework. Algorithm 8, shows an example API with these two fields defined. Table 5.1, lists the current capabilities of the framework and describes their functionality—the top-level routes the instruction and data packet to either MHA-Selector or the RT-MHA module.

Algorithm 8 API Example

```
{   "method": "set_parameters",
    "data":{
        "binaural": {
            "beamformer":{
                "bf_on_off": true,
                ...
            }
            ...
        },
        "left": {
            "wdrc_0":{
                "g50": $i_0$,
                "g80": $j_0$,
                ...
            }
            ...
            "wdrc_n":{
                "g50": $i_n$,
                "g80": $j_n$,
                ...
            }
            ...
        },
        "right": {
            "wdrc":{
                "g50": [$i_0$, ..., $i_n$],
                "g80": [$j_0$, ..., $j_n$],
                ...
            }
            ...
        }
        ...
    }
}
```

Table 5.1: Describes the OSP-Net JSON request and response behavior.

Method	**Data**	**Response**
get_mha	N/A	Sends back the current MHA selected
set_mha	JSON object containing ”mha_selection” Tells OSP which MHA to use	”success” - Current MHA has been changed ”failed at ...” - Was not able to change MHA
get_parameters	N/A	Sends back all of the parameters that are externally available
set_parameters	JSON containing the parameters that the user is requesting to update	”success” - Everything has been updated ”failed at ...” - Was not able to update the parameter
get_info	N/A	Sends back the JSON object composed of: + ”mha_seector” - JSON object describing all of MHA available + ”current_mha” - JSON object describing all of the library components of the running MHA

MHA-Selector

The MHA-Selector is an endpoint, and it either responds to the three commands. The first command is ”*get_mha*,” which returns the currently running MHA Design's index. Next command is ”*set_mha*”, which changes the currently running MHA design based on the requested ”*mha_selection*”. The final command is ”*get_info*,” which replies with all the types of MHA Designs available on the device. Algorithm 7 best describes the types of responses the MHA Selector gives when it receives each of these commands.

RT-MHA Module

The RT-MHA module routes the ”*method*” and ”*data*” to the library modules that compose the MHA design. The JSON data structure reflects how data gets routed in OSP. Each library module first gets grouped into one of three groups, ”left,” ”right,” or ”binaural,” based on its function defined by the MHA designer. The next level of routing uses the name of the library module. The MHA designer sets the name and plays a crucial role in structuring the API. The names of each library module should be unique, so there are no conflicts. If there are conflicts between two library modules, then any parameters within those two library modules with the same names will share the same value. This side effect may be a feature that an MHA designer may want to exploit, but we only recommend it for the most experienced user.

We noticed that designers would implement arrays of the same library modules in many MHA designs. Therefore, we felt that grouping the modules into an array was necessary since it made the API more readable and much more condensed. We designed a mechanism called the Array Manager, which uses the library module names to automatically make the API more readable. To use the Array Manager, the MHA designer needs to label all of the modules that make up the array with the same name with each library module's name ending in "[*x*]," where *x* is a number representing the index of the library module in this array. Then the Array Manager will handle the packing and unpacking all the parameters into an array.

An MHA designer will have to define each library module's name and the grouping of the library module in the constructor of the MHA design. Algorithm 9 shows an example MHA design constructor, which has a one-to-one mapping to the example API in Algorithm 8. One thing of note is that the difference between the left and right channels is how we named the wdrc library modules. The right channel is named using the convention that the array manager can handle. Therefore, the right channel's API is represented as an array and is much more compact.

Library Module

The library modules are endpoints where we parse the rest of the JSON data structure and extract the parameters required. Each library module must inherit the abstract class called parameter, see Algorithm 10. This class requires the library module to have four key components: a "name" used as a way to address this module; a "set_param" function that parses the incoming "data" JSON object; a "get_param" function that constructs a JSON object of the current parameter values to send to the requester; and a "get_info" function that constructs a JSON object full of information about the library module. Signal processing researchers, who will be primarily developing library components, must follow two conventions to comply with our Rapid API. First, the "name" variable needs to be a part of the library module's constructor so that the

Algorithm 9 Example MHA Design Constructor

```
/***********Binaural Channel*****************/
beamforming = new beamformer("beamformer", ...);
parameters["binaural"].push_back(beamforming);
//Initialize other binaural library modules

/***********Left Channel*****************/
wdrcLeft[0] = new wdrc("wdrc_0", ...);
parameters["left"].push_back(wdrcLeft[0]);
...
wdrcLeft[n] = new wdrc("wdrc_n", ...);
parameters["left"].push_back(wdrcLeft[n]);
//Initialize other left library modules

/***********Right Channel*****************/
wdrcRight[0] = new wdrc("wdrc[0]", ...);
parameters["right"].push_back(wdrcRight[0]);
...
wdrcRight[n] = new wdrc("wdrc[n]", ...);
parameters["right"].push_back(wdrcRight[n]);
//Initialize other right library modules
```

MHA designers can assign the library module a name. It is similar to how a router assigns an endpoint its address. The other convention is that a parameter name ending with ”_r” is a read-only parameter, meaning that the ”set” function will not parse and write to that parameter. This convention needs to be enforced by the library module developer and gives the developer greater freedom in designing the control access to the parameters.

5.2.2 Hot Swapping Mechanism

In Algorithm 7, we describe the MHA swapping mechanism in ”*set_mha*” function. The way the MHA swapping mechanism works is that it saves the parameters of the current MHA design. Then the mechanism unloads the current MHA design before loading the next MHA design. Unfortunately, the ”*set_mha*” function is not real-time compatible, which means that during a hot-swap from one MHA design to another, it will most likely create clicks and pops in the audio system. These audio artifacts will be uncomfortable for the end-user. Therefore,

Algorithm 10 Parameter Abstraction

```cpp
using json = nlohmann::json;
class parameter {
public:
    explicit parameter(std::string name_): name(std::move(name_)){};
    virtual ~parameter() = default;
    virtual void set_param(json j) = 0;
    virtual void get_param(json &j) = 0;
    virtual void get_info(json &j) = 0;
    std::string get_name(){return name;}

protected:
    std::string name;
};
```

we designed a mechanism that eliminates these artifacts by temporarily bypassing RT-MHA by passing audio from the microphone to the speaker. In order to make the transition less jarring, the mechanism takes 10μs to transition from the MHA design to the pass-thru system. Similarly, once the new MHA design is loaded, the mechanism transitions from the pass-thru to the MHA design over 10μs.

5.2.3 Evaluation

Here we will evaluate the real-time performance of the swapping mechanism and measure the output behavior of the ear level assembly during the transition period.

Setup: In this experiment, we ran the hearing aid algorithm described in Figure 4.2 with the beamformer disabled in two different configurations. The first configuration uses the single-core partitioning scheme, and the second is the domain-based partitioning scheme that utilizes all three real-time CPU resources. The OSP device was connected to an external computer over a WiFi connection. The external computer will send a request to switch between the two MHA designs every 10 seconds. We ran the experiment once with a background task preventing CPU idling to isolate the effects of the hot-swapping mechanism. The experiment was run for 600 seconds, and we recorded the time it takes for the firm real-time task to process a data buffer.

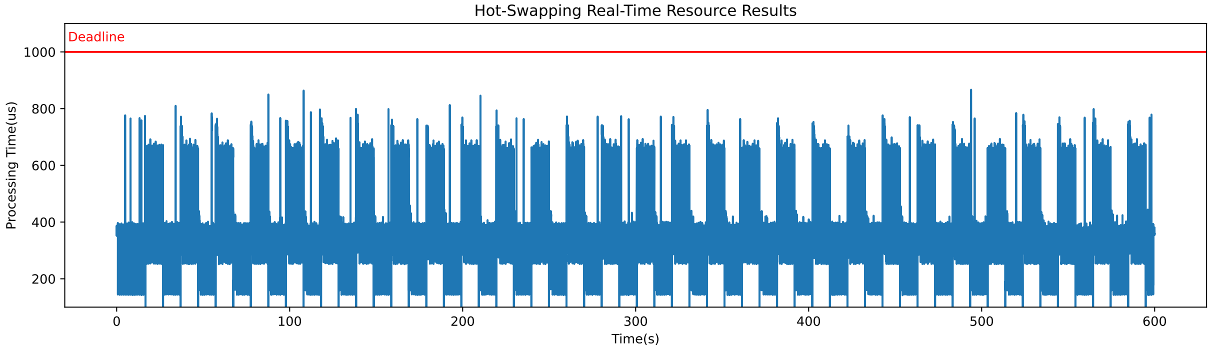


Figure 5.4: Describes the results of hot-swapping mechanism which switches between two MHA designs every 10 seconds for 600 seconds.

Results: The data in Figure 5.4 shows that the hot-swapping mechanism is real-time safe. Over 10 minutes, we toggled between two MHA designs 60 times and never missed the deadline. Upon further inspection, we do see that there are tangible impacts of switching between MHA designs. Right after the transition is completed, there seems to be a period where there is a temporary increase in processing time. This increase in processing time is likely due to the time it takes for the speculating mechanisms to understand the algorithmic and memory patterns change. Therefore, this increase in processing time must be accounted for when designing the MHA algorithms if it were to be used in experiments that toggle between multiple them.

5.3 Live OScope: Debugger for OSP

Developing and implementing signal processing and master hearing aid (MHA) algorithms on any research tools come with challenges, especially when deployed on an embedded system. The main challenge is the lack of visibility in most hearing aid research tools once the algorithms are deployed in an embedded system. In our experience, the most challenging part is understanding why an algorithm does not work in the real world even though it was tested thoroughly in a simulated environment. Therefore, we feel that it is essential for these research tools to incorporate an intuitive interface for debugging these algorithms. Therefore, we extended the Open Speech Platform (OSP) framework to include a debugging tool called the Live OScope.

We were inspired by the oscilloscope, an analog tool used to debug signals when we designed the Live OScope feature of the OSP framework. The Live OScope allows algorithm developers to insert test points for monitoring during run time. The Live OScope feature has two components: the recorder and the display. The recorder is a part of the OSP framework; it continuously records the data from the test points into a circular buffer of a predefined length. The recording happens until the display component asks for the data. At which time the data gets streamed from the buffer in its entirety to the display component. After the streaming, the recorder component starts sampling the data again. The display component, an external application running on a remote device, displays the waveforms captured on the embedded device. The display component can refresh at a set interval like an oscilloscope and save the waveform captures.

We first will technically describe the Live OScope mechanism. Then we will evaluate the feature's impact on the real-time performance of the embedded system. Finally, we will describe the utilities of this tool through four case studies that utilized this tool.

5.3.1 Mechanics of Live OScope

Figure 5.5 describes the Live OScope mechanism at a high-level. There are two parts to the mechanisms the recorder and the display. The recorder allows the MHA designer to insert test points along the signal processing chain at any sampling rate. These test points are designed to record every sample of data in a circular buffer of fixed lengths with a power of 2. By making the circular buffer lengths power of 2, the buffers have the least impact on the real-time processing. However, this means that the buffer size increases in steps of power of 2. Each test point is addressed via a unique name. For example in Figure 5.5, the unique names are ”Left After AFC” and ”Output Left.”

The central part of the recording mechanism exists on a separate thread running on the soft real-time core. This thread creates a TCP server that waits for instruction from the client.

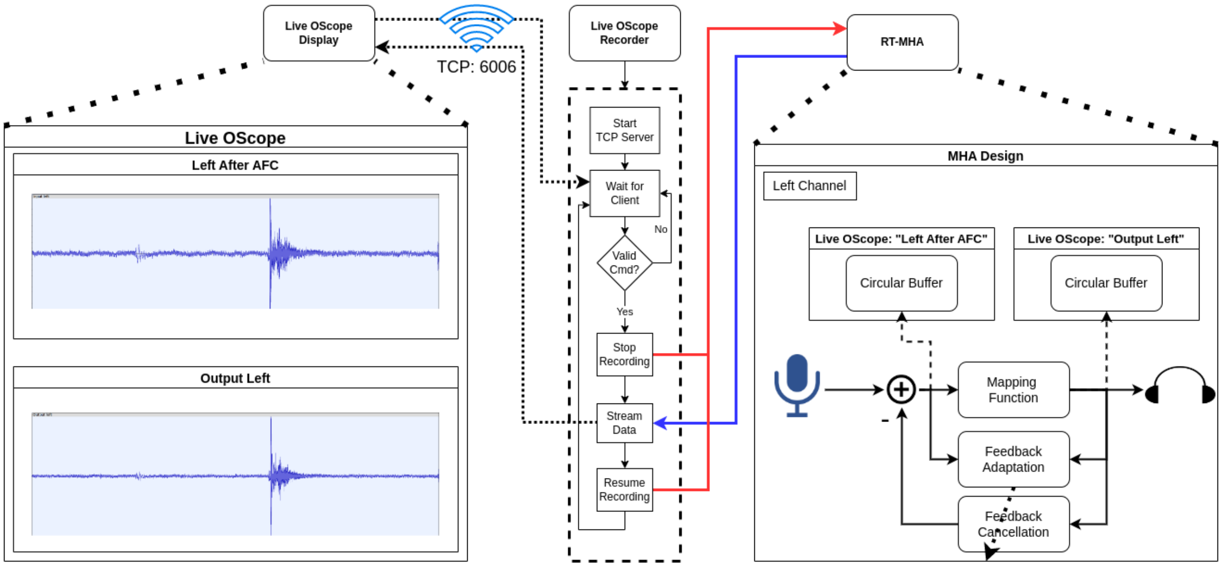


Figure 5.5: Compilation flow for the proposed domain specific language.

The client, in this case, is the Live OScope display located on an external device, like a laptop. The client requests data from the server by asking for all the data from the available test points or just data from one test point by referring to it via the unique name. Once the server gets that request, it stops the circular buffer from recording while it transmits the requested data. After the data has been transferred, the circular buffer records the incoming data samples.

Now the display mechanism can use this data to show the waveform on the external device. If the display requests all the test points, then the display mechanism will receive a separate data packet with each test point's unique name and sampling rate for each test point. On the other hand, if the display mechanism only requests one test point, it will only receive one such packet.

5.3.2 Use Cases

The researcher will have access to the raw data and can use it to create other applications using the foundations of the display mechanism. Here we will discuss four such use cases for the Live OScope mechanisms to show its potential as a research tool for the hearing aid community.

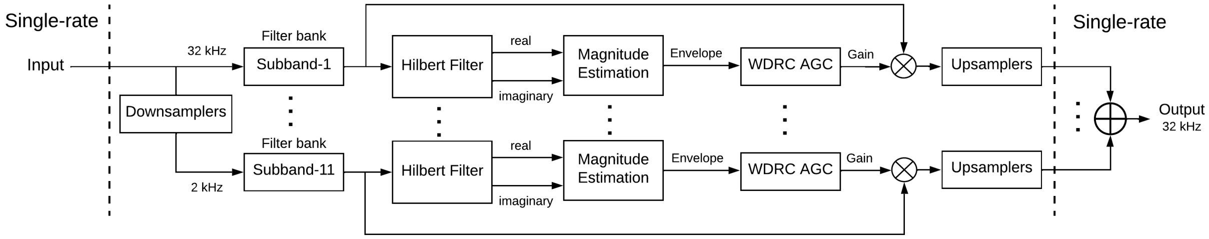


Figure 5.6: A block diagram of our multirate multiband hearing aid amplification system. A 32 kHz input signal is separated into eleven frequency channels with different sampling rates. Each channel is then individually processed. Lastly, all bands are brought back to the original sampling rate and combined to create the output signal for real-time playback.

Debugger

The primary application for the Live OScope is that of a debugging tool. Run-time issues are one of the hardest to fix, especially when there is limited visibility into the system. Therefore, we designed Live OScope after oscilloscopes, which are used to probe physical wires that connect different circuit components. The researcher will use the Live OScope to probe the data lines that connect different signal processing algorithms to give them better visibility into the system. For example, while developing a novel 11-band multi-rate mapping function in Figure 5.6 [SSH$^+$22], we ran into an issue that only occurred in the code running on the OSP device. While testing the new function in a simulated environment, all the different components functioned as per specification. However, testing it on the OSP device, we discovered that the mapping function was not functioning correctly, mainly at lower frequencies. Since this issue was the only observable symptom without the Live OScope, we could only speculate that the issue was caused due to the down-sampling module. That reasoning turned out to be false. Once we used the Live OScope to record all of the data lines in Figure 5.6, we found that the issue was an incorrectly configured buffer in the Hilbert Filter.

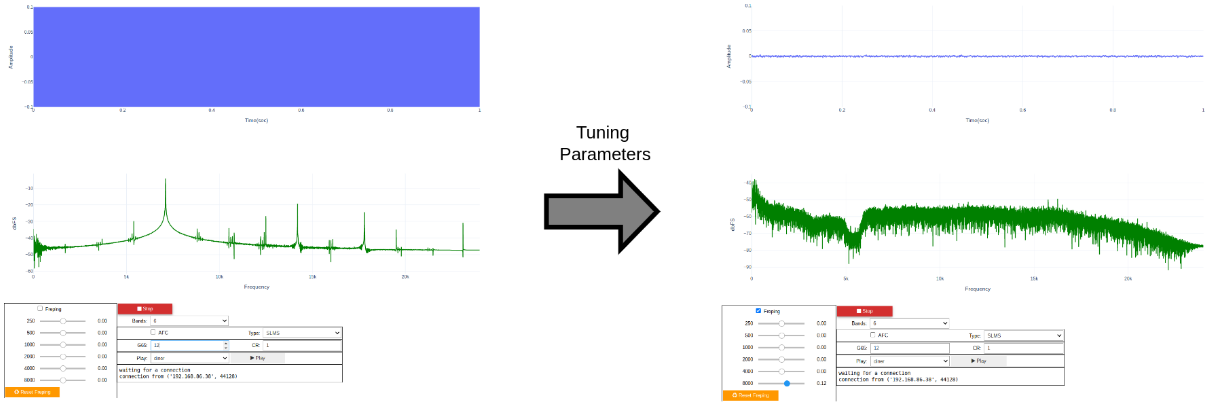
Calibration

A hearing aid is calibrated on a hearing aid measurement device, like the Verifit by Audioscan. Audiologists and clinical researchers expect that the adjustments they make to a patient's hearing aids in the software are accurately reflected in the real world. Therefore, the algorithm and the devices must be well-calibrated. We would need to calibrate a hearing aid device because microphones and receivers are all unique in how they interpret sounds. However, the algorithm is expecting a standardized input and output. Therefore, a calibrating algorithm is needed to interpret the data produced by the microphones and receivers so that the hearing aid algorithm can work as designed.

During the development of the OSP device, we discovered that calibrating a hearing aid device is a very cumbersome process. Usually, the manufacturer calibrates a hearing aid before it ships to the end-user. However, OSP is an open-source platform, meaning we will not have control over the types of devices the software runs on. Therefore, we needed to have the capability to calibrate the hearing aid algorithm using a hearing aid measurement device. This calibration is usually done through many measurements of the input and output characteristics. We found that the best way to calibrate the device when we do not have visibility into the hearing aid algorithm is by measuring the effects of the WDRC curve for every frequency band. This way of calibration requires at least four measurement points per frequency band to reproduce the WDRC curve. On the other hand, Live OScope allows us to monitor the magnitude estimation algorithm for each frequency band, e.g., see Figure 5.6, which means we only need to measure once per frequency band to derive the required calibration. The incorporation of Live OScope reduced the work required by the researcher to calibrate these devices in-situ by at least four times.

Visualization Tool

Live OScope enables research to visualize the impacts of the algorithm in-situ. This capability allows researchers to study the impacts of the algorithm in the field without the need



☐ Freping

250	0.00
500	0.00
1000	0.00
2000	0.00
4000	0.00
8000	0.00

Bands: 6
☐ AFC Type: SLMS
G65: 12 CR: 1
Play: diner

```
waiting for a connection
connection from ('192.168.86.38', 44128)
```

☑ Freping

250	0.00
500	0.00
1000	0.00
2000	0.00
4000	0.00
8000	0.12

Bands: 6
☐ AFC Type: SLMS
G65: 12 CR: 1
Play: diner

```
waiting for a connection
connection from ('192.168.86.38', 44128)
```

Figure 5.7: Showcases how Live Oscope can be used to tune parameters in the hearing aid, while worn on a person. This is an example of using frequency warping, a.k.a. freping, to attenuate acoustic feedback. The blue graph is the raw signal from the Live OScop and the green graph is the frequency spectrum of the blue graph.

for external measurement devices. For example, [LCR+19] showed that theoretically, Frequency Warping, a.k.a. freping, can be used as an alternative to adaptive feedback cancelation. We implemented the freping algorithm in OSP and, using the Live OScope feature, showed the capability in-situ. Figure 5.7 shows freping in action while being worn on a person. The spikes in the green graph on the left show the acoustic feedback when freping is turned off, and the lack of the spikes on the right shows that feedback was eliminated when we tuned the freping parameter on the 8000 Hz frequency band. Using Live OScope, we tuned the parameters until the acoustic feedback was eliminated and now can study the algorithm's effect in many different environments. This demo validated the theoretical results of Lee et al.'s work [LCR+19].

Real-Time Reinforcement Learning

One of the critical areas of hearing aid research is finding the best parameters for configuring a given mapping function. This personalized configuration is known as the prescription of the hearing aid. Current best practices include an audiologist taking a pure tone audiometry (PTA) of the hearing aid user. Then they put the results of the PTA through a prescription software that returns a prescription for the user's hearing aid. This prescription is not personalized to the individual; instead, it is a generic prescription representing the average of a group of people with similar PTAs. Therefore, the audiologist will further tune the prescription based on subjective feedback from the hearing aid user. This process usually lasts over a few months since the user should try their hearing aid's prescription in multiple environments before fine-tuning it.

Researchers are looking for algorithmic ways to adjust the hearing aid user's prescription based on their feedback. However, one of the critical issues with adjusting based on user preference is that the feedback might be non-convex, meaning they may not have just one preferred prescription per environment. In [BRM22] the authors ran a self-adjustment study to identify whether individual users could repeatably select their preferred prescription. Their study concluded that hearing aid users can repeatably select the prescription on average to within ±4dB of their mean. This result indicates a range of preferences, which varies by users, even in a controlled environment.

One such algorithm which has shown promise in parameter tuning using feedback rather than a loss function has been reinforcement learning (RL). Using a simulated environment, we could show that given an objective function, e.g., PESQ, as a surrogate to human feedback, we could use an algorithm called multi-armed bandit, the most straightforward RL algorithm, to tune the hearing aid for a better fit. However, the results we gathered were based on a convex objective function that does not accurately represent the behaviors of a human being.

For that reason, we used the Live OScope mechanism to integrate OSP into the Gym.AI framework, the most widely used RL framework. This integration will allow researchers to test such algorithms in the field rapidly. An RL algorithm requires two pieces of information, State and Reward, and returns an Action to the environment. In our case, the State is derived from data that is gathered through Live OScope, the Reward is the feedback from the hearing aid user, and the Action is the parameter adjustment of the hearing aid device, Figure 5.8. This integration will allow researchers to rapidly prototype novel RL algorithms for hearing aids in the lab and the field.

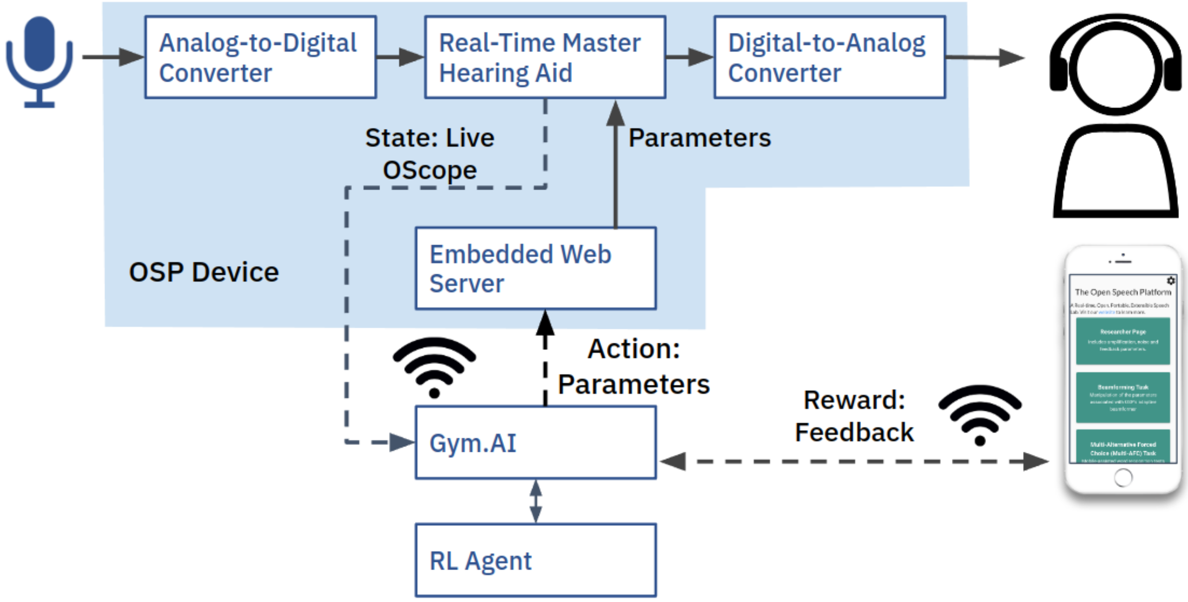


Figure 5.8: Shows an example of how we are able to integrate Live OScope with the Gym.AI framework to enable research in reinforcement learning.

5.3.3 Evaluation

All debugging mechanisms have some impact on the performance of the system. Therefore, in this evaluation, we evaluate the impact of the Live OScope mechanism on real-time performance. Also, we evaluate the best case transfer speed to determine the system's capabilities.

Setup: For all evaluations, we will have an OSP device running the MHA design from Figure 4.2. The OSP device will act as a WiFi hotspot; the only device connected to it will be a laptop running Linux. The OSP device will contain the Live OScope recorder, and the laptop will contain the Live OScope display.

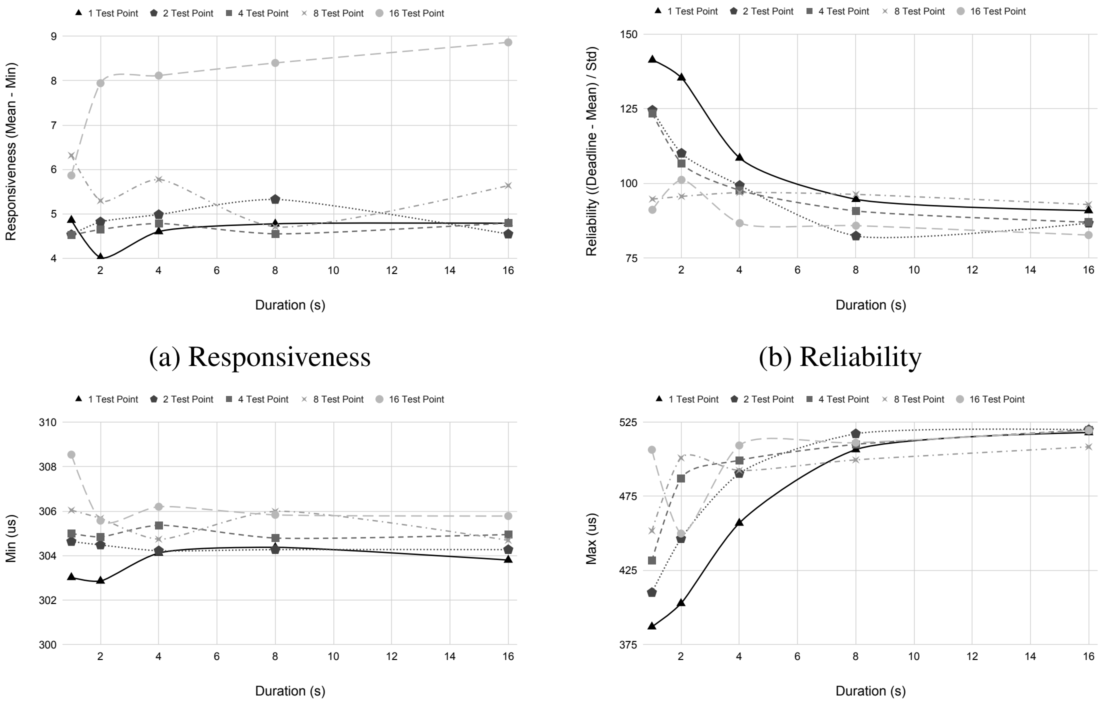
In order to best understand the impact of the Live OScope mechanism, we will be evaluating the setup along the two parameters the researcher would have control over. The first is the duration of the recording length in seconds ranging from 1 second to 16 seconds. The other parameter is the number of testing points ranging from 1 test point to 16 test points. We will vary both of these parameters while keeping the sampling rate at 32 kHz and evaluate the parameter's impact on the real-time performance and the best case transfer speed. For both evaluations, we will place the OSP device around 2 cm away from the laptop.

Results: Figure 5.9 best describes the real-time overhead incurred by the Live OScope mechanism. The graphs of the responsiveness, reliability and maximum computation time display a similar characteristic. The graphs all degrade with the increase in the recordings' duration and the number of test points. The pattern in which the performance degrades suggests that it is due to the amount of memory usage. Looking closer, we can see a limit to the overhead. In our example, the real-time overhead is around 150μs based on the maximum computation time. Digging deeper, we can see that in the raw real-time data, Figure 5.10, the overhead occurs only when the recording mechanism transfers the data from the firm real-time core to the soft real-time core to transmit it to the display mechanism. The last thing of note from Figure 5.5 is that the minimum computation time is not affected by memory usage but is purely impacted by the number of test points. This behavior is because only the function calls for each test point impact the computation time when the record location is pre-loaded in the CPU's cache.

The second part of this evaluation tries to determine the limitation of Live OScope. Figure 5.11 describes the result of the transfer speed test across the different configurations of Live OScope. The results show that the transfer speed is not affected by the different configurations. Upon further analysis, we discovered that the limitation of the Live OScope mechanism is the WiFi link.

5.4 Discussion

This chapter described and evaluated two mechanisms we designed to address the bottlenecks in designing MHA algorithms for the OSP framework. The first mechanism enables hot-swapping between MHA designs to address the issue of scalability of the RT-MHA framework. The second mechanism, called Live OScope, provides a debug tool that gives researchers a view into the algorithm running on the device in-situ.



(a) Responsiveness

(b) Reliability

(c) Minimum Computation Time

(d) Maximum Computation Time

Figure 5.9: Describes the real-time performance of the RT-MHA system as a function of the number of test points used and the duration of the recording.

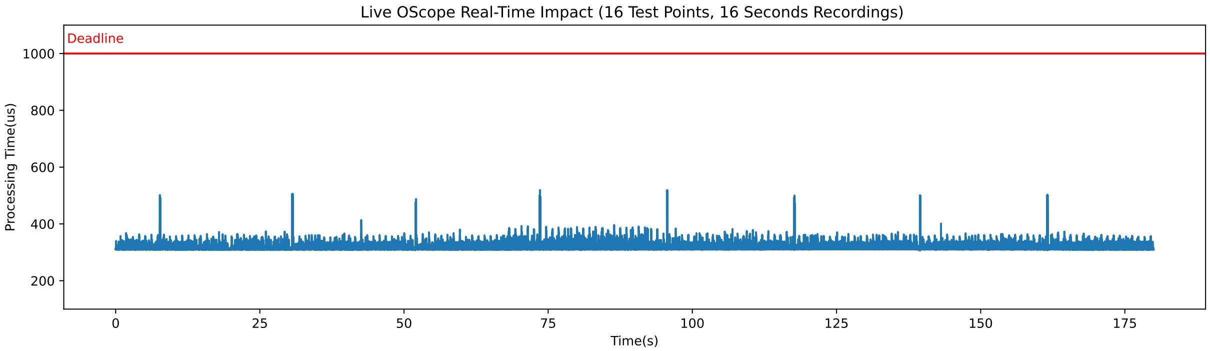


Figure 5.10: Describes the real-time impact of Live OScope when the display mechanism requests data from the recording mechansim. The configuration of the Live OScope is 16 Test Points with 16 seconds of data recorded. The spikes only occur when the data is requested due to data migrating from the caches in the firm real-time CPU to the soft real-time CPU.

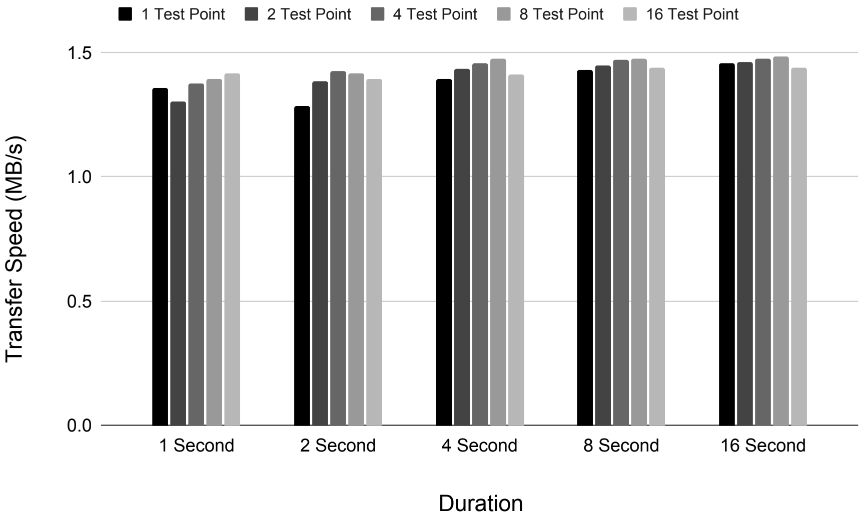


Figure 5.11: Results from the transfer speed test showing that the Live OScope mechanism is WiFi link speed bound.

During our evaluations, we showed that when the mechanism is active but not invoked, they have little to no impact on the operation of the RT-MHA framework. However, when invoking the mechanism, there is a temporary increase in computation time, which call the real-time overhead. For both mechanisms, the real-time overhead is the highest when dealing with reconfiguring the memory. In the hot-swapping mechanism, the overhead is the highest when the mechanism loads the newly allocated MHA design into RT-MHA. This real-time overhead is due to the cache learning the new data patterns associated with the recently loaded MHA design. For the Live OScope, the highest real-time overhead occurs when the Live OScope displays requests for the data. At this time, the data is transferred from the firm real-time core to the soft real-time core, and this is what causes the spike in the real-time performance. Based on our evaluation, both mechanisms seem to have an upper limit on the real-time overhead. Therefore, the MHA designers must consider these upper limits when designing algorithms with these mechanisms. The upper limits must be determined as they change based on the hardware and operating environment.

6 Conclusion

In this dissertation we set out to prove that mobile computing devices are suitable for hearing aid research. We did this by building a proof-of-concept prototype on the Qualcomm 410c microprocessor. Objectively we showed that the device performed just as well as commercial off-the-shelve hearing aids. Also, subjectively we showed that study participants are willing to use such devices for extended field studies.

We identified the best configuration of the operating environment to get the best real-time performance while sacrificing the least in programmability. We found that Linux with the FIFO real-time scheduler and partitioning the CPU resources between firm real-time and soft real-time using IsolCPU created the ideal operating environment for the hearing aid research tool. Also, we discovered that the idle mechanism in these mobile devices heavily impacts the system's real-time performance, meaning that there is a trade-off between real-time performance and energy efficiency that needs to be considered per the hearing aid algorithm.

Given the partitioned CPU resources, we explore the best ways to distribute the workload. We designed three distribution techniques: a manual partition scheme based on domain knowledge and two automated partitioning schemes using directed acyclic graphs. We evaluated each partitioning scheme and found that the best partitioning method is the one that is the least efficient at packing tasks within the real-time computation resource. This partitioning makes the ideal trade-off between real-time performance and energy efficiency.

In order to enable researchers to compare and contrast multiple hearing aid designs, we extended our framework to enable hot-swapping. To accomplish the goal of hot-swapping, we first designed a hearing aid algorithm selector. After which, we created a standardized system called Rapid API to enable researchers to create an API for their hearing aid design. These APIs enable the interaction between the hearing aid algorithms and the external applications. Lastly, we designed and evaluated the hot-swapping mechanism.

The final mechanism we designed and evaluated in this dissertation is the Live OScope, a tool inspired by the oscilloscope used for debugging signals in the hearing aid algorithm. We evaluated this mechanism and showed that the impact of such a mechanism is minimal. The Live OScope is far more than just a debugging tool and opens up new research possibilities that did not exist prior, e.g., a mechanism through which we can perform live reinforcement learning on an edge computing node.

Bibliography

[All87] Steve T Allworth. *Introduction to real-time software design*. Macmillan International Higher Education, 1987.

[BKM$^+$19] Arthur Boothroyd, Christine Kirsch, Carol Mackersie, Shaelyn Painter, and Harinath Garudadri. Usability assessment of a wearable speech-processing platform. *The Journal of the Acoustical Society of America*, 146(4):2878–2878, 2019.

[BM17] Arthur Boothroyd and Carol Mackersie. A "goldilocks" approach to hearing-aid self-fitting: User interactions. *American journal of audiology*, 26(3S):430–435, 2017.

[BRM22] Arthur Boothroyd, Jennifer Retana, and Carol L Mackersie. Amplification self-adjustment: Controls and repeatability. *Ear and Hearing*, 43(3):808–821, 2022.

[DGM$^+$08] Simon Doclo, Sharon Gannot, Marc Moonen, Ann Spriet, Simon Haykin, and KJ Ray Liu. Acoustic beamforming for hearing aid applications. *Handbook on array processing and sensor networks*, pages 269–302, 2008.

[Dou15] Timur Doumler. Cppcon 2015: Timur doumler "c++ in the audio industry", Oct 2015.

[Ele09] Knowles Electronics. RVA-90020-NXX datasheet, 2009.

[Ele14] Knowles Electronics. RVA-90080-NXX datasheet, 2014.

[FBS$^+$20] Francesco Fraternali, Bharathan Balaji, Dhiman Sengupta, Dezhi Hong, and Rajesh K Gupta. Ember: energy management of batteryless event detection sensors with deep reinforcement learning. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 503–516, 2020.

[Fri19] Jeff Friesen. Introducing json. In *Java XML and JSON*, pages 187–203. Springer, 2019.

[GBL$^+$17] Harinath Garudadri, Arthur Boothroyd, Ching-Hua Lee, Swaroop Gadiyaram, Justyn Bell, Dhiman Sengupta, Sean Hamilton, Krishna Chaithanya Vastare, Rajesh Gupta, and Bhaskar D Rao. A realtime, open-source speech-processing platform for research in hearing loss compensation. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 1900–1904. IEEE, 2017.

[GCBM18] Tobias Goehring, Josie L Chapman, Stefan Bleeck, and Jessica JM Monaghan*. Tolerable delay for speech production and perception: Effects of hearing ability and experience with hearing aids. *International Journal of Audiology*, 57(1):61–68, 2018.

[Haw20] Joseph E. Hawkins. human ear, Oct 2020.

[HKL+17] Tobias Herzke, Hendrik Kayser, Frasher Loshaj, Giso Grimm, and Volker Hohmann. Open signal processing software platform for hearing aid research (openmha). In *Proceedings of the Linux Audio Conference*, pages 35–42, Stanford University, California, US, 2017. CCRMA.

[HSG18] Sean Hamilton, Dhiman Sengupta, and Rajesh Gupta. Introducing automatic time stamping (ats) with a reference implementation in swift. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 138–141. IEEE, 2018.

[JKG21] Charlotte T Jespersen, Brent C Kirkwood, and Jennifer Groth. Increasing the effectiveness of hearing aid directional microphones. In *Seminars in Hearing*, volume 42, pages 224–236. Thieme Medical Publishers, Inc., 2021.

[Jor16] Lindsey E Jorgensen. Verification and validation of hearing aids: Opportunity not an obstacle. *Journal of otology*, 11(2):57–62, 2016.

[Kat05] James M Kates. Principles of digital dynamic-range compression. *Trends in amplification*, 9(2):45–76, 2005.

[Kat08] James M Kates. *Digital hearing aids*. Plural publishing, 2008.

[Kat19] James M Kates. Limitations of the envelope difference index as a metric for nonlinear distortion in hearing aids. *Ear and hearing*, 2019.

[Kay16] Jackie Kay. Introduction to real-time systems, Jan 2016.

[KBS+18] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, and Yuvraj Agarwal. Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation. In *Proceedings of the 5th Conference on Systems for Built Environments*, pages 11–20, 2018.

[KDF+11] Gitte Keidser, Harvey Dillon, Matthew Flax, Teresa Ching, and Scott Brewer. The nal-nl2 prescription procedure. *Audiology research*, 1(1), 2011.

[KHM^{+}19] Hendrik Kayser, Tobias Herzke, Paul Maanen, Chaslav Pavlovic, and Volker Hohmann. Open master hearing aid (openmha)—an integrated platform for hearing aid research. *The Journal of the Acoustical Society of America*, 146(4):2879–2879, 2019.

[KHM^{+}22] Hendrik Kayser, Tobias Herzke, Paul Maanen, Max Zimmermann, Giso Grimm, and Volker Hohmann. Open community platform for hearing aid algorithm research: open master hearing aid (openmha). *SoftwareX*, 17:100953, 2022.

[KLHL09] Gibak Kim, Yang Lu, Yi Hu, and Philipos C Loizou. An algorithm that improves speech intelligibility in noise for normal-hearing listeners. *The Journal of the Acoustical Society of America*, 126(3):1486–1494, 2009.

[KP17] Nasser Kehtarnavaz and Issa M Panahi. Smartphones as research platform for hearing improvement studies. *The Journal of the Acoustical Society of America*, 141(5):3495–3495, 2017.

[kSLB^{+}16] Yoon kyu Sung, Lingsheng Li, Caitlin Blake, Josh Betz, and Frank R. Lin. Association of hearing loss and loneliness in older adults. *Journal of Aging and Health*, 28(6):979–994, 2016. PMID: 26597841.

[KVdBMW07] Thomas J Klasen, Tim Van den Bogaert, Marc Moonen, and Jan Wouters. Binaural noise reduction algorithms for hearing aids that preserve interaural time delay cues. *IEEE Transactions on Signal Processing*, 55(4):1579–1585, 2007.

[Lab20] Statistical Signal Processing Research Laboratory. Research and development platform - ssprl - the university of texas at dallas. Available at https://www.utdallas.edu/ssprl/hearing-aid-project/research-platform (2020/01/15), Jan 2020.

[LCR^{+}19] Ching-Hua Lee, Kuan-Lin Chen, Bhaskar D Rao, Harinath Garudadri, et al. On mitigating acoustic feedback in hearing aids with frequency warping by all-pass networks. In *Interspeech*, volume 2019, page 4245. NIH Public Access, 2019.

[LCZ^{+}19] Mingchao Liang, Kuan-Lin Chen, Wenyu Zhang, Ching-Hua Lee, Bhaskar D Rao, and Harinath Garudadri. Noise managment features in open speech platform. *The Journal of the Acoustical Society of America*, 146(4):2916–2916, 2019.

[Loh22] Niels Lohmann. JSON for Modern C++, 1 2022.

[LRG17] Ching-Hua Lee, Bhaskar D Rao, and Harinath Garudadri. Sparsity promoting lms for adaptive feedback cancellation. In *Proc. Europ. Signal Process. Conf.(EUSIPCO)*, pages 226–230, 2017.

[MAF19] David W Maidment, Yasmin HK Ali, and Melanie A Ferguson. Applying the com-b model to assess the usability of smartphone-connected listening devices in adults with hearing loss. *Journal of the American Academy of Audiology*, 30(5):417–430, 2019.

[MBG18] Carol Mackersie, Arthur Boothroyd, and Harinath Garudadri. Research on hearing-aid self-adjustment by adults. *The Journal of the Acoustical Society of America*, 143(3):1743–1743, 2018.

[MBG20] Carol L Mackersie, Arthur Boothroyd, and Harinath Garudadri. Hearing aid self-adjustment: Effects of formal speech-perception test and noise. *Trends in hearing*, 24:2331216520930545, 2020.

[MBL19] Carol Mackersie, Arthur Boothroyd, and Alexandra Lithgow. A "goldilocks" approach to hearing aid self-fitting: Ear-canal output and speech intelligibility index. *Ear and hearing*, 40(1):107–115, 2019.

[MD14a] Roger L Miller and Amy Donahue. Open speech signal processing platform workshop, Oct 2014.

[MD14b] Roger L Miller and Amy Donahue. Open speech signal processing platform workshop, October 2014.

[ME17] Joe Mario and Jeremy Eder. Low latency performance tuning for red hat enterprise linux 7, 2017.

[MK90] H Gustav Muller and Mead C. Killion. An easy method for calculating the articulation index. *Hearing Journal*, 43:14–17, 1990.

[NIH17] NIH. Hearing loss and hearing aid use (infographic), Dec 2017.

[O$^+$17] World Health Organization et al. *Global costs of unaddressed hearing loss and cost-effectiveness of interventions: a WHO report, 2017*. World Health Organization, 2017.

[PHS$^+$18] Louis Pisha, Sean Hamilton, Dhiman Sengupta, Ching-Hua Lee, Krishna Chaithanya Vastare, Tamara Zubatiy, Sergio Luna, Cagri Yalcin, Alex Grant, Rajesh Gupta, et al. A wearable platform for research in augmented hearing. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 223–227. IEEE, 2018.

[PWZ$^+$19] Louis Pisha, Julian Warchall, Tamara Zubatiy, Sean Hamilton, Ching-Hua Lee, Ganz Chockalingam, Patrick P. Mercier, Rajesh Gupta, Bhaskar D. Rao, and Harinath Garudadri. A wearable, extensible, open-source platform for hearing healthcare research. *IEEE Access*, 7:162083–162101, 2019.

[RAK$^+$19] Varsha Rallapalli, Melinda Anderson, James Kates, Lauren Balmert, Lynn Sirow, Kathryn Arehart, and Pamela Souza. Quantifying the range of signal modification in clinically fit hearing aids. *Ear and hearing*, 2019.

[Ram19] Diana Ramos. *Exploring IVSHMEM in the Jailhouse Hypervisor*. PhD thesis, ISEP, 2019.

[RHG+18] Michael Risoud, J-N Hanson, Fanny Gauvrit, C Renard, P-E Lemesre, N-X Bonne, and Christophe Vincent. Sound source localization. *European annals of otorhinolaryngology, head and neck diseases*, 135(4):259–264, 2018.

[SG21] Alice Szymanski and Zachary Geiger. *Anatomy, Head and Neck, Ear*. StatPearls Publishing, Treasure Island (FL), 2021.

[SHG13] Nicolas Serrano, Josune Hernantes, and Gorka Gallardo. Mobile web apps. *IEEE software*, 30(5):22–27, 2013.

[SM99] Michael A Stone and Brian CJ Moore. Tolerable hearing aid delays. i. estimation of limits imposed by the auditory path alone using simulated hearing losses. *Ear and Hearing*, 20(3):182–192, 1999.

[SM02] Michael A Stone and Brian CJ Moore. Tolerable hearing aid delays. ii. estimation of limits imposed during speech production. *Ear and Hearing*, 23(4):325–338, 2002.

[SM03] Michael A Stone and Brian CJ Moore. Tolerable hearing aid delays. iii. effects on speech production and perception of across-frequency variation in delay. *Ear and Hearing*, 24(2):175–183, 2003.

[SM05] Michael A Stone and Brian CJ Moore. Tolerable hearing-aid delays: Iv. effects on subjective disturbance during speech production by hearing-impaired subjects. *Ear and Hearing*, 26(2):225–235, 2005.

[SMMD08] Michael A Stone, Brian CJ Moore, Katrin Meisenbacher, and Ralph P Derleth. Tolerable hearing aid delays. v. estimation of limits for open canal fittings. *Ear and Hearing*, 29(4):601–617, 2008.

[SSC+21] Alice Sokolova, Dhiman Sengupta, Kuan-Lin Chen, Rajesh Gupta, Baris Aksanli, Fredric Harris, and Harinath Garudadri. Multirate audiometric filter bank for hearing aid devices. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, pages 1436–1442. IEEE, 2021.

[SSH+22] Alice Sokolova, Dhiman Sengupta, Martin Hunt, Rajesh Gupta, Baris Aksanli, Fredric Harris, and Harinath Garudadri. Real-time multirate multiband amplification for hearing aids. *IEEE Access*, 2022.

[SZH+20] Dhiman Sengupta, Tamara Zubatiy, Sean K. Hamilton, Arthur Boothroyd, Cagri Yalcin, Dezhi Hong, Rajesh Gupta, and Harinath Garudadri. Open speech platform: Democratizing hearing aid research. In *Proceedings of the 14th EAI InternationalConference on Pervasive Computing Technologies for Healthcare*, 2020.

[TGM+18] Maike AS Tahden, Anja Gieseler, Markus Meis, Kirsten C Wagener, and Hans Colonius. What keeps older adults with hearing impairment from adopting hearing aids? *Trends in hearing*, 22:2331216518809737, 2018.

[Tym20] Tympan. Tympan. Available at https://shop.tympan.org (2020/01/15), 2020.

[V+20] Trevor Charles Vannoy et al. *Enabling rapid prototyping of audio signal processing systems using system-on-chip field programmable gate arrays*. PhD thesis, Montana State University-Bozeman, Norm Asbjornson College of Engineering, 2020.

[VSS+17] Lukeshwari Verma, Himanshu Kumar Sanju, Bibina Scaria, Mayank Awasthi, Aparna Ravichandran, Ashritha Kaki, and Savalam Gnana Rathna Prakash. A comparative study on hearing aid benefits of digital hearing aid use (bte) from six months to two years. *International archives of otorhinolaryngology*, 21(03):224–231, 2017.

[VV10] Dennis Van Vliet. Practicing audiology long-distance. *The Hearing Journal*, 63(2):60, 2010.

[VVG91] Michael Valente, Maureen Valente, and Joel Goebel. Reliability and intersubject variability of the real ear unaided response. *Ear and hearing*, 12(3), 1991.

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220. ACM, 2013.

[YC21] Zhijian Yang and Romit Roy Choudhury. Personalizing head related transfer functions for earables. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 137–150, 2021.

[YS16] Jim Yen and Dhiman Sengupta. Long distance time transfer using time reversal (t3r). In *Proceedings of the 47th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 99–106, 2016.