A Comparison of Open-Source Linux Frameworks for an Augmented Musical Instrument Implementation

Eduardo A. L. Meneses
Input Devices and Music Interaction Lab/CIRMMT
McGill University, Canada
eduardo.meneses@mail.mcgill.ca

Johnty Wang
Input Devices and Music Interaction Lab/CIRMMT
McGill University, Canada
johnty.wang@mail.mcgill.ca

Sérgio Freire
School of Music
Universidade Federal de Minas Gerais (UFMG), Brazil
sfreire@musica.ufmg.br

Marcelo M. Wanderley
Input Devices and Music Interaction Lab/CIRMMT
McGill University, Canada
marcelo.wanderley@mcgill.ca

ABSTRACT

The increasing availability of accessible sensor technologies, single board computers, and prototyping platforms have resulted in a growing number of frameworks explicitly geared towards the design and construction of Digital and Augmented Musical Instruments. Developing such instruments can be facilitated by choosing the most suitable framework for each project. In the process of selecting a framework for implementing an augmented guitar instrument, we have tested three Linux-based open-source platforms that have been designed for real-time sensor interfacing, audio processing, and synthesis. Factors such as acquisition latency, workload measurements, documentation, and software implementation are compared and discussed to determine the suitability of each environment for our particular project.

Author Keywords

Augmented Instruments, Sensor Interfaces, Latency

CCS Concepts

•**Hardware** → **Sensor devices and platforms;** •**Applied computing** → *Sound and music computing; Performing arts;*

1. INTRODUCTION

The increasing availability of accessible sensor technologies, single board computers, and prototyping platforms have resulted in a growing number of frameworks explicitly geared towards the design and construction of Digital Musical Instruments (DMIs) and Augmented Musical Instruments (AMIs).

New challenges arise from these new tools, such as the limitations imposed even on the implementation of simple sensing techniques [1], or the choice of the proper hardware and data acquisition method.

Moreover, we need to consider that all available options carry constraints, and no tool can be used blindly in every project. This paper presents a number of existing platforms that were considered when building a new prototype for our specific AMI (The GuitarAMI), and the process of identifying and evaluating some key factors that were used to influence our choice. First, we present the goals and requirements of our project, followed by the description and performance comparisons of three selected frameworks: Bela, Prynth and a custom Sound Processing Unit (SPU).

2. REQUIREMENTS OF THE GUITARAMI PROJECT

The GuitarAMI is an Augmented Musical Instrument (AMI) in development at the Input Devices and Interaction Laboratory (IDMIL) at McGill University [2]. The next step for GuitarAMI research is an evaluation of its use in performance and composition. The current prototype uses SuperCollider (SC) for mapping and sound synthesis and requires audio input/output, wireless data input from a sensor interface using Open Sound Control (OSC) through Wi-Fi, five switches, and a small LCD for simple visual feedback.

Wi-Fi capability, specifically the ability to send and receive OSC messages wirelessly, is considered an essential requirement for the GuitarAMI due to the modular design, allowing multiple modules in different guitars or even other instruments to interact among themselves using the GuitarAMI as an access point[1].

To provide visual feedback we chose to use a small embedded display. The current communication interfaces available for those devices are Serial Peripheral Interface (SPI), and Inter-Integrated Circuit (I^2C). Therefore, it is expected that the framework provides us at least one of the mentioned communication interfaces.

Even though the GuitarAMI provides visual feedback through a small display, we expect the AMI to be used in headless mode, i.e., without the need of a monitor or keyboard. The performer should be able to simply connect the audio cable and turn on the instrument for operation.

Therefore, to fulfill the GuitarAMI's current requirements, the selected framework needs to provide the following capabilities: 1) Ability to run SuperCollider, and enough processing power to run the GuitarAMI code in real time; 2) Wi-Fi connectivity; 3) Provide five analog/digital sensor inputs (to implement five switches); 4) Provide SPI or I^2C communication interfaces; 5) Ability to run in headless

[1]One example of multiple instruments interaction using the GuitarAMI can be seen at `https://youtu.be/n-r9-c2jAG0`.

NIME'19, June 3-6, 2019, Porto Alegre, Brazil

mode; and 6) Fulfil the audio/data latency requirements discussed below.

Apart from the requirements above, another important specification is the audio and data latency. As discussed by McPherson, Jack, and Moro [3], latency perception depends on the musical and visual context. The classical guitar sound envelope, similarly to other plucked string instruments, consists of quick attack time and short sustain. This prominent attack may suggest that the instrument's perceivable latency thresholds should be closer to the percussion instruments.

However, qualitative latency perception tests for monitor systems in live performances using stage monitors show that latency under 6.5 ms is considered ideal, while latency values up to 14.5 ms are considered fair [4]. For the classical guitar case, we can then consider Wessel and Wright's suggestion of a maximum latency of 10 ms [5] as a reasonable objective, due to its proximity to the average between the maximum ideal and fair latency values presented by Lester and Boley [4].

Finally, time variation (jitter) is another essential factor to be considered in signal acquisition. In this particular scenario, the absence of jitter is critical for the audio input/output, while data acquisition may accept some variation regarding both "continuous" signals and switch sensors used to change modes of operation. In the latter case, we consider a higher tolerance, in the order of tens of milliseconds, similar to the sensitivity stated by Mäki-Patola and Hämäläinen for theremin control [6].

The GuitarAMI's SuperCollider code requires a minimum block size of 128 samples to avoid X-runs[2] using any of the tested frameworks. This block size is the smallest buffer that allows computing the audio output within the real-time constraints and, any block smaller than 128 samples produces enough X-runs on both the BeagleBone Black and the Raspberry Pi (the single-board computers used in the tested frameworks) to compromise audio quality.

We expect to work with a minimal sample rate of 44.1kHz for ADC/DAC to ensure proper audio quality, although our approach will be to choose a sample rate that produces the smallest latency for each framework. We also expect to work with the smallest number of periods/buffer possible (2 periods/buffer), and highest analog sample rate for data acquisition (non-audio ADC) for each tested framework, both parameters are set to reduce overall latency.

Apart from the measurements presented in section 4, there are also qualitative aspects to consider when choosing a framework for a particular project. Those aspects may relate with particular needs or workflow and methodology, i.e., the necessity to evaluate or modify the code in real-time, or the easiness in working with a particular protocol such as I^2C or SPI. Out-of-the-box functionalities required by the project are also desirable, i.e., providing high-quality audio inputs or built-in wireless capabilities.

3. SELECTED PLATFORMS

Among the available microcontroller and single-board computer platforms, there are frameworks explicitly created to support artistic creation. We can highlight Bela and Prynth as two platforms created as DMI oriented frameworks. Additionally, we also tested a custom sound process unit.

3.1 Bela

Bela is a framework for creating audio and interactive applications, developed and maintained by the Augmented Instruments Laboratory at Queen Mary University of London (QMUL) [8, 9]. The initial research evolved into an open source platform that was launched by a successful crowdfunding campaign and is currently maintained by the Augmented Instruments Laboratory at Queen Mary University of London (QMUL). More information about Bela can be found on the official website[3], and the code can be found on Github[4].

The Bela hardware consists of a BeagleBone Black and a custom "cape" (add-on board) that provides audio, and analog/digital input/output functionalities. The software environment is a Linux distribution "running audio at higher priority than the Linux kernel" [8]. Bela can run code compiled from C++, Csound, and SuperCollider as well as Pure-data patches. The framework is highly customized to provide the DMI designer a plug-and-play experience, and the Bela Integrated Development Environment (IDE) exposes parameters commonly modified during instrument design.

3.2 Prynth

Prynth is designed as a programmable sound synthesizer running on single-board computers. Franco and Wanderley first published this development in [10]. Similarly to Bela, Prynth evolved into an open source platform, and information about the framework can be found on the official website[5]. Hardware and software specifications can be found on GitHub[6]. The project is currently maintained by IDMIL, at McGill University.

The Prynth hardware consists of a Raspberry Pi, and a custom "hat" (add-on board) hosting a Teensy microcontroller for analog and digital I/O functionalities. Prynth's framework does not provide any audio functionality out-of-the-box. Instead, the platform presents a pre-configured JACK API that allows the use of any audio interface in the system. Prynth's software is a custom headless Linux distribution that provides an IDE capable of running SC code. However, Prynth software aims to not only to execute SC code but also serve as a coding environment similar to the standard SC IDE, allowing the user to evaluate portions of the code and send commands to the SC language. Since the Prynth framework does not provide an audio interface by default, in our tests we employed a Fe-Pi Audio board[7].

3.3 Custom Sound Process Unit (SPU)

Another viable alternative to Bela and Prynth is to use the frameworks' building blocks to create a custom solution. This approach provides a customization level similar to the *Satellite CCRMA* [11] and can be attractive if the desired design requires a specific software or hardware initially unsupported by those frameworks. Thus we built a Sound Processing Unit (SPU) suited to our needs using a Raspberry Pi 3 B+ and a Fe-Pi Audio board. Building a custom system allowed us to run SuperCollider and a plugin host simultaneously, controlling the system remotely through a VNC connection. Even though this approach provides more flexibility, the drawback is that this solution will not take advantage of the optimization found in the aforementioned frameworks.

[2] An X-run is a large denomination for processing errors caused either by buffer underruns or a buffer overruns. X-runs happens when an audio application is not fast enough to either send or process data to/from the ALSA audio buffer, usually causing audible sound artifacts [7].

[3] `http://bela.io/`.
[4] `https://github.com/BelaPlatform/Bela`.
[5] `https://prynth.github.io/`.
[6] `https://github.com/prynth/prynth`.
[7] `Https://fe-pi.com/products/fe-pi-audio-z-v2`.

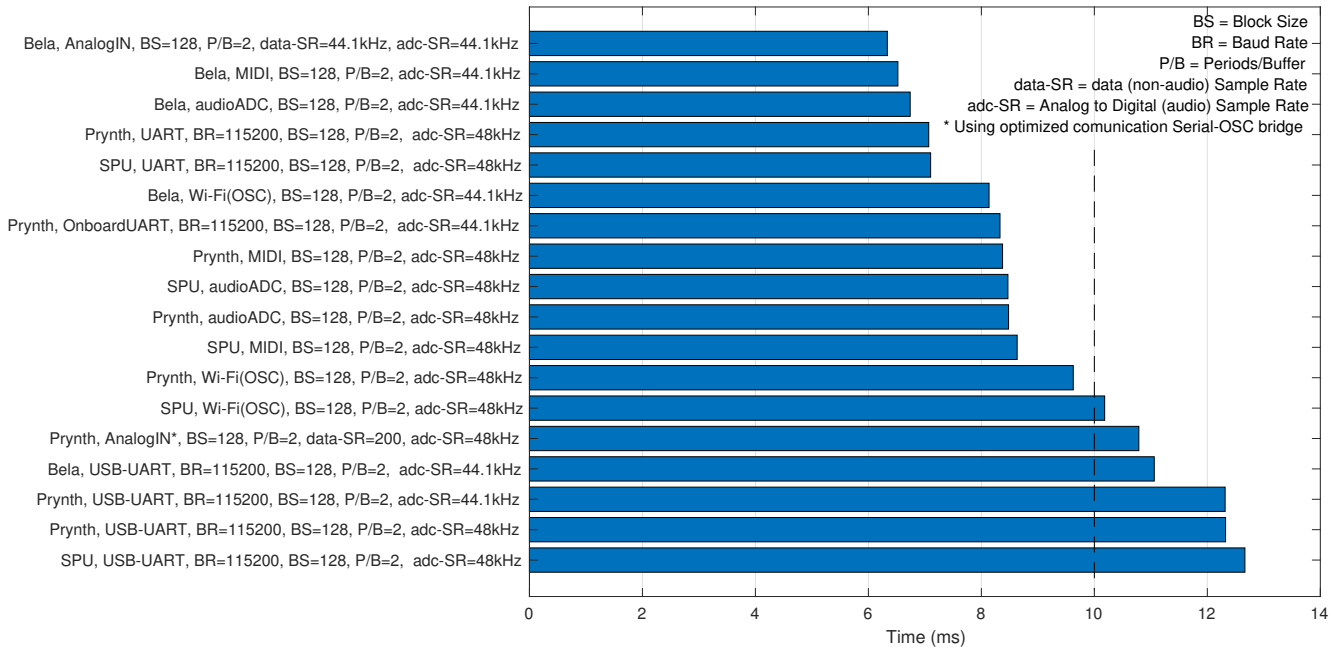

Figure 1: Overall results for latency tests using different frameworks, inputs and configurations. The presented values are average results of 1000 tests, and the input triggering the Latency Jig could be audio (audioADC) or non-audio: AnalogIN, USB-UART, UART, OnboardUART (using a secondary microcontroller), MIDI over USB, and OSC though Wi-Fi.

4. PERFORMANCE COMPARISONS

4.1 Latency Measurement Setup

We used the latency measurement system from a previously published study [3]. In this system, a microcontroller based testing device is used to measure the time interval between the output of a trigger (toggling a pin in the microcontroller from low to high) and the detection (by the same microcontroller) of an audio signal sparked by this trigger in an external device. The measurement device continuously outputs measured latency values via a serial console so that the results can be stored and analyzed later. We constructed our version of the measurement hardware, henceforth referred to as the "Latency Jig", according to the original specifications described in the prior work.

The device under test, in this case, consists of a combination of the sensor input hardware, input drivers, synthesis software as well as the audio output hardware. Therefore, the resultant latency value reflects the combined performance of all parts used to implement a particular system for a specific platform under consideration.

According to the requirements and restrictions presented in section 2, we performed framework latency measurements using different input methods and sample rates for audio and data. Along with the use of audio input (AudioADC) to measure end to end latency, the following methods were applied to test latency using non-audio triggers: 1) framework analog pins—AnalogIN, 2) serial communication through USB—USB-UART[8], 3) serial communication using built-in analog pins—UART, 4) serial communication using the framework's additional microcontroller—OnboardUART, 5) MIDI over USB—MIDI, and 6) OSC over UDP protocol through Wi-Fi—Wi-Fi(OSC).

We executed 1000 measurements for each configuration, and the average results can be seen in Figure 1. Those measurements are described in more details in the next sessions.

[8]Universal Asynchronous Receiver/Transmitter (UART) is the hardware responsible for transmitting and receiving serial data.

4.2 Audio Configuration

Initial latency measurements using Bela's non-audio ADC reproduced similar results in comparison with the previously published data found in [3]. We then proceeded to test Bela, Prynth, and SPU in three distinct end-to-end tests: 1) Using analog input to trigger an audio response; 2) Triggering an ESP32 microcontroller to send an OSC message using UDP protocol and consequently triggering an audio response; and 3) Using the analog trigger as audio input and send the received signal as an audio response.

It is important to state that all three platforms have exposed parameters for audio and data configuration. The Bela IDE allows the user to choose the block size (vector size), the sample rate for audio and non-audio ADC/DAC, the number of analog/digital channels, and the gain for audio ADC/DAC. Prynth software allows the user to choose the OSC message receiver in SuperCollider (sclang or scsynth), the block size, the sample rate for audio and non-audio ADC/DAC, each sensor status, the number of periods for the JACK Audio Connection Kit, filters for each sensor, and Wi-fi SSID/password. Since the SPU uses a standard Linux distribution, it is possible to change any parameter for the JACK Audio Connection Kit and sensor acquisition, although it is necessary to change parameters individually in each system component, while Bela and Prynth provide a simplified interface for real-time configuration.

According to the requirements and considerations stated in section 2, the frameworks were configured as follows:

- Bela: BlockSize=128, Periods/Buffer=2, Analog Sample Rate=22.05kHz, Audio Sample Rate=44.1kHz;
- Prynth: BlockSize=128, Periods/Buffer=2, Analog Sample Rate=200Hz, Audio Sample Rate=48kHz; and
- SPU: BlockSize=128, Periods/Buffer=2, Serial(UART) BaudRate=115200, Audio Sample Rate=48kHz.

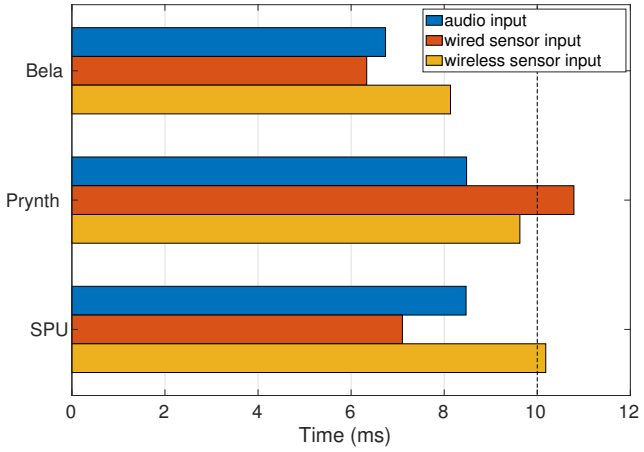

Figure 2: Average results of 1000 framework latency measurements using the best configuration for each platform.

4.3 Audio-Audio Latency

We measured end-to-end audio latency using the Latency Jig trigger output as an audio signal, and the Device Under Test (DUT) audio output as a return signal. For all three platforms, we measured good latency results (under 10ms), and virtually no jitter. For the configuration above, Bela presented an average audio end-to-end latency of 6.74 ms, while Prynth and the SPU obtained 8.48 ms and 8.47 ms, respectively.

4.4 Wired Sensor Input

For the wired sensor acquisition, we tested the serial (UART) interface sending raw bits, and USB sending both raw bits or MIDI. Even though MIDI over USB yielded good latency values and virtually no jitter, the best results for Bela and Prynth were achieved using each framework's preset analog inputs configurable through Bela's IDE or Prynth Software. Since the Raspberry Pi does not have analog inputs, some alternatives for analog data acquisition for the SPU includes using analog to digital converters such as the MCP3008, or use USB-based microcontrollers such as Teeny or Arduino. The best results were achieved using the serial (UART) interface sending raw bits directly to SuperCollider. Bela obtained a data to audio end-to-end average latency of 6.33ms, SPU's latency had an average of 7.1ms, and Prynth, 10.79ms. While Bela exhibited virtually no jitter in data to audio tests and SPU had a small variation of $\Delta_{lat} \approx 2.9ms$, Prynth showed a jitter value of $\Delta_{lat} \approx 16ms$.

4.5 Wireless Sensor Input

As discussed in section 2, we used Wi-fi for wireless communication. While the Raspberry Pi was configured as an access point in the Prynth and SPU implementations, the BeagleBone Black does not have a built-in wireless LAN adapter[9]. To provide Wi-Fi functionality for Bela, we added a wireless USB network adapter (Linksys WUSB600N ver.2). For end-to-end data to audio latency using OSC over Wi-fi, Bela presented an average measurement of 8.14 ms, Prynth's latency has an average of 9.63 ms, and SPU presented 10.18 ms latency. Even though the average latency is acceptable, all three frameworks exhibited considerable jitter. The measured jitter values were $\Delta_{lat} \approx 16.31ms$ for Bela, and $\Delta_{lat} \approx 18.7ms$ for Prynth / SPU.

[9]While the BeagleBone Wireless exists, it does not fully work with Bela due to conflicts with the analog inputs.

The best results for each framework can be seen in Figure 2. These results are presented as an average of 1000 measurements performed for each setup.

4.6 Latency Analysis

Even though the measured average latency and the full variability (Δ_{lat}) can be useful information to predict if the system performance is acceptable, it can also lead to false expectations regarding performance. Therefore, we look at two additional metrics: the Standard Deviation (SD) and Empirical Cumulative Distribution Function (ECDF) of the latency values.

4.6.1 Standard Deviation

Table 1: Average latency and standard deviation for each framework.

Acquisition mode	Average latency (ms)	Standard deviation (ms)	Coefficient of variation (%)
Bela			
audio	6.74	0.021	0.31
wired	6.34	0.015	0.24
wireless	8.14	1.165	14.31
Prynth			
audio	8.48	0.004	0.05
wired	10.79	3.122	28.93
wireless	9.63	2.005	20.82
SPU			
audio	8.47	0.002	0.02
wired	7.10	0.304	4.28
wireless	10.18	1.835	18.02

Table 1 presents the Standard Deviation (SD) alongside the average latency values, as well as a coefficient of variation which computes the ratio of the SD and the average latency. Empirically, a lower coefficient of variation (less than 1%) suggests that there is negligible variability (jitter). The audio acquisition modes, as expected due to the strict demands of real-time processing, exhibits as expected, minimal jitter in all frameworks. We can also see that the wired input methods generally perform considerably better than the wireless counterparts, except in the case of Prynth.

4.6.2 Empirical Cumulative Distribution Function

Compared to box and whisker plots or histograms, the Empirical Cumulative Distribution Function (ECDF) is an interesting way to present latency values, since the horizontal (time) locations of large vertical (probability) steps show where discrete groupings of similar latency values lie. Plotting the latency values as an ECDF in Figures 3 and 4 for the wired and wireless cases respectively reveals a distinctive staircase-like characteristic, which clearly shows that, for all configurations, about 90% of values fell under the 10ms threshold. The location of the horizontal axis "steps" also reveal the relatively discrete distribution of the latency values, which are likely a result of the underlying processor scheduler in a multi-threaded environment or, in the case of Wi-Fi, transmission intervals. Further analysis with tools such as Wireshark may help confirm this behavior.

Specific to the wired tests (Figure 3), we can observe a high probability of experiencing latencies slightly lower than, although very close to the average. Even though the number of measurements found above the higher concentration marks cannot be interpreted as outliers—they do not constitute single occurrences but values periodically present— they are dispersed enough not to cause a noticeable effect on performance.

For the wireless acquisition latency tests (Figure 4), we find a more diffuse measurement distribution, with at least

two major convergence points and several small steps evenly distributed along the time axis. It is interesting to note that this equidistant distribution appears in all frameworks, and the performance is similar regardless of the platform. The most prominent steps found in Figure 4 occur at $\approx 6.75ms$, $\approx 9.5ms$, and $\approx 12ms$.

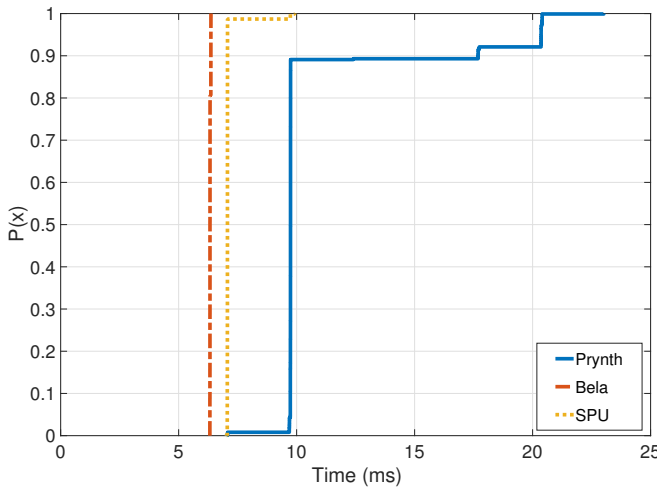

Figure 3: Empirical cumulative distribution function of Prynth, Bela, and SPU using wired sensor acquisition. Vertical line projections in the X axis represent the value around we found higher concentration of measurements.

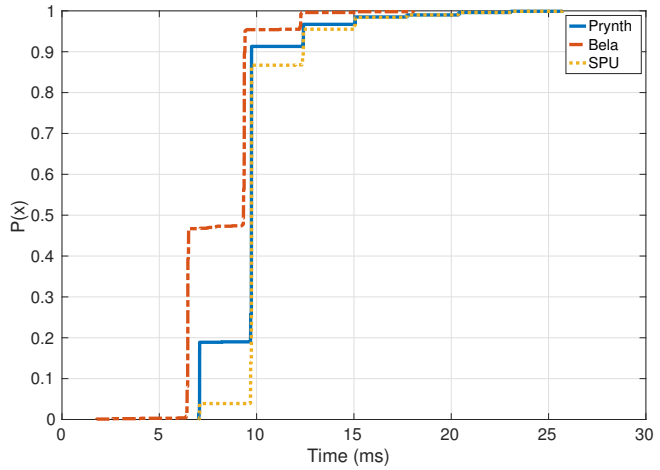

Figure 4: Empirical cumulative distribution function of Prynth, Bela, and SPU using wireless sensor acquisition. Similarly to Figure 3, vertical line projections in the X axis represent the value around we found higher concentration of measurements.

4.7 CPU Load

One last metric is related to the CPU load and, more importantly, the load averages on each platform. We monitored the platforms during the latency measurement tests for at least 15 minutes to obtain load averages.

The CPU usage value is a just-in-time value of how busy is the CPU, while the load averages measure the CPU "traffic", i.e., the number of tasks waiting to be processed. It is interesting to note the difference between the two metrics since the load average is not only affected by CPU usage, but also by tasks with an uninterruptible state (leading to disk access) [12]. That may lead to apparently strange situations where, despite low CPU usage, the load average indicates high CPU waiting time.

The table 2 presents the CPU usage (%) and the load averages for each platform. For comparison, the CPU usage is normalized between 0 and 100%, while the load averages were normalized using the number of cores of each system. These normalizations imply that 100% CPU usage is equivalent to full use of all available cores in the system. Load averages lower than 1 imply that there is no process waiting in line and all incoming requests are being processed immediately, while values higher than 1 suggest that some CPU requests have to wait for processing.

Table 2: CPU usage (normalized between 0 and 100%) and load average (normalized by the number of cores) for each framework.

			Load Averages		
Framework	**Cores**	**CPU**	1 min	5 min	15 min
Prynth	4	17.2%	0.03	0.05	0.33
Bela	1	23.4%	1.91	1.84	1.88
SPU	4	35.5%	0.51	0.43	0.31

It is important to state that, even though it is possible to get the specific CPU % for the SuperCollider application (sclang and scsynth), we present the load averages that reflect all active tasks. We choose this value since it better estimates the amount of additional processing headroom available when the entire system is running. Each framework was set using the latest available SD card images (for Bela and Prynth) at the time of the experiment, and the only modification was Wi-Fi configuration for both systems. The SPU use Raspbian 8 (Jessie), and during the latency tests the only active software were SuperCollider (sclang, scsynth, and scide), and Claudia[10].

5. DISCUSSION

All tested frameworks are suitable for the given application. However, none of them poses as a perfect solution in this particular scenario.

Bela presents the best average latency and jitter results, as well as high-quality audio inputs and outputs out-of-the-box. However, it cannot evaluate partial SC code (even though it can be used to write the code).

Prynth presents acceptable average latency results, built-in configurable filters for analog sensors, and the ability to evaluate partial SC codes. However, it requires an external audio interface and presents higher jitter results. Even though Prynth's wired sensor average latency is above the established limit, the connected switches are used in GuitarAMI only to set operation modes, and for this task the measured latency is acceptable.

The SPU also presents acceptable average latency results, flexibility to run several audio applications patched trough JACK, and the ability to evaluate partial SC codes. However, the SPU requires manual adjust for all configurations and connections.

5.1 Framework design decisions

All tested frameworks required a fair amount of customization to fulfill all the GuitarAMI requirements, which cannot

[10]Claudia is a LADISH frontend, responsible for loading/saving audio connections and launching audio software automatically (`https://kx.studio/Applications:Claudia`).

be considered a drawback per se. Since the frameworks are intended to be flexible and employable in several scenarios, customizations are expected in any project. Decisions taken by the creators and maintainers of the frameworks have an enormous impact on the strengths and trade-offs of each platform. For instance, when Bela creators chose the BeagleBone Black as their single-board computer, they also imposed Wi-Fi restrictions due to BeagleBone's lack of a built-in Wi-Fi adapter. Similarly, Prynth's creators opted not to embed any audio interface in their board, requiring the user to include an audio board. It is evident that both platform creators deliberately made choices to strengthen their systems while choosing acceptable trade-offs. The design choices of Bela and Prynth resulted in some similar functionalities such as the browser-based IDE, but also in very distinct features, e.g., Prynth's use of the Teensy and multiplexers to receive and process analog sensor signals. It is the responsibility of the user to check all the strengths and drawbacks of each system according to a specific project.

5.2 Documentation

As stated previously, the chosen framework will probably require adaptation for each project, in which case users may benefit from proper documentation. Documentation poses a particular challenge in any development project, but it is particularly challenging for open-source projects.

Indeed, most of the customization made in both platforms required specific knowledge of their software architectures and, in most cases, the best location to obtain information was the framework's users forum. Both Prynth and Bela have active forums that allow users and platform designers to interact, which provides the required support and allows users to search for their particular problems in older posts. We can assume that for Prynth and Bela the documentation is composed not only of the main documentation website (Wiki) and discussion forums, but also the documentation of the framework components (e.g., BeagleBone Black documentation for Bela, or Raspberry Pi / Teensy documentation for Prynth and SPU).

Ideally, a quick search should reveal the strengths and weaknesses of each framework, allowing the user to choose the most suitable system, i.e., the framework that will require the least amount of customization to achieve a particular project objective. However, in practice, designers often find themselves in a situation where it is necessary to acquire a certain level of expertise in several frameworks to take a more informed decision about the ideal platform for each particular situation. Hence, the availability of documentation, for both platform usage/implementation and performance metrics, then becomes just as important as the actual implementation itself, They work in tandem to avoid "reinventing the wheel". More information will often engage more users, thereby increasing community size and support, and also generating a positive feedback loop. From our experience, both the Bela and Prynth communities could benefit immensely in this regard.

6. CONCLUSION

The selection of an embedded environment for the implementation of an augmented guitar instrument required a comparison of different available platforms. In this paper, we presented the results of our latency and workload tests on Bela, Prynth, and the custom-built SPU, three Linux-based open-source platforms that have been designed for real-time sensor interfacing, audio processing, and synthesis.

Using audio and data acquisition methods, as required for building the augmented instrument, the test data could be analyzed and we can conclude that, even though all systems can be employed in our project, each of them carry advantages and drawbacks related to the software access and exposed settings, documentation, compatibility, and amount of customization required.

The GuitarAMI prototypes built using Bela, Prynth, and the SPU will be further evaluated in future performances and composition projects that will yield further insight on the use of these frameworks.

7. REFERENCES

[1] Carolina B. Medeiros. *Advanced Instrumentation and Sensor Fusion Methods in Input Devices for Musical Expression*. PhD thesis, McGill University, Montreal, Canada, 2015.

[2] Eduardo A. L. Meneses, Sérgio Freire, and Marcelo M. Wanderley. GuitarAMI and GuiaRT: two independent yet complementary augmented nylon guitar projects. In *Proc. International Conference on New Interfaces for Musical Expression (NIME)*, pages 222–227, Blacksburg, USA, 2018.

[3] Andrew McPherson, Robert Jack, and Giulio Moro. Action-sound latency: Are our tools fast enough? In *Proc. International Conference on New Interfaces for Musical Expression (NIME)*, pages 20–25, Brisbane, Australia, 2016.

[4] Michael Lester and Jon Boley. The effects of latency on live sound monitoring. In *Proc. Audio Engineering Society (AES) 123th Convention*, New York, USA, 2007.

[5] David Wessel and Matthew Wright. Problems and prospects for intimate musical control of computers. *Computer Music Journal*, 26(3):11–22, 2002.

[6] Teemu Mäki-Patola and Perttu Hämäläinen. Latency tolerance for gesture controlled continuous sound instrument without tactile feedback. In *Proc. International Computer Music Conference (ICMC)*, Coral Gables, USA, 2004.

[7] Mark Constable. The unofficial ALSA wiki: Independent ALSA and Linux audio support site. `https://alsa.opensrc.org/`. Accessed: 2018-12-06.

[8] Andrew P. McPherson and Victor Zappi. An environment for submillisecond-latency audio and sensor processing on BeagleBone Black. In *Proc. Audio Engineering Society (AES) 138th Convention*, Warsaw, Poland, 2015.

[9] Liam Donovan, S. M. Astrid Bin, Jack Armitage, and Andrew P. McPherson. Building an ide for an embedded system using web technologies. In *Web Audio Conference*, London, UK, 2017.

[10] Ivan Franco and Marcelo M. Wanderley. Prynth: A framework for self-contained digital music instruments. In *Proc. 12th International Symposium on Computer Music Multidisciplinary Research (CMMR)*, pages 357–370, São Paulo, Brazil, 2016.

[11] Edgar Berdahl, Spencer Salazar, and Myles Borins. Embedded networking and hardware-accelerated graphics with Satellite CCRMA. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 325–330, Daejeon, Republic of Korea, May 2013. Graduate School of Culture Technology, KAIST.

[12] Brendan Gregg. Linux load averages: Solving the mystery. `http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html`, 2017. Accessed: 2018-12-11.