FPGA-accelerated Real-Time Audio in Pure Data

Clemens Wegener
Interface Design Group
Bauhaus-Universität Weimar
clemens.wegener@uni-weimar.de

Sebastian Stang
The Center for Haptic Audio Interaction Research (CHAIR)
sebastian@chair.audio

Max Neupert
The Center for Haptic Audio Interaction Research (CHAIR)
max@chair.audio

ABSTRACT

With the advent of fast ARM processors more audio products are running embedded Linux systems and using high level languages to implement real-time signal processing. Still, embedded Linux systems can't compete with the processing power of desktop computers to implement complex signal processing as needed for physical modeling algorithms. This paper describes how to interface custom digital logic circuits in an Field Programmable Gate Array (FPGA) with the Linux operating system to speed up processing. The hardware used is a Terasic DE10-Nano development kit equipped with an Intel [1] Cyclone V SoC. [2] We are running Pure Data (Pd) on Linux and communicating with a mass-interaction network for physical modeling sound synthesis on the FPGA. The code referenced in this publication is available online. [3]

1. INTRODUCTION

1.1 DSP on embedded Platforms—a brief Overview

The landscape of available hardware platforms capable of running digital audio applications is diverse and ranges from microcontrollers to full Linux systems running on multi core CPUs. To pick the right platform for a project is an optimization challenge to get the most computing power for as little resources as possible. But not only that: In most use-cases additional specifications are important. A fan-less, passively cooled system is usually required and—if used as an instrument—it's mandatory that latency and jitter of the input to output pipeline stays in tolerable [1] limits. For light projects, platforms without the overhead of an operating system have the advantage to be affordable, compact, and robust. The Axoloti [2], Electrosmith Daisy, and Teensy [3] boards are examples of this category. When it comes to compact Linux boards, the choices are almost endless. Leaving industrial single board computers and SOMs [4] aside, community friendly options are mostly ARM architectures which include the Pine64 or Odroid boards. The Raspberry Pi is in this group and is arguably the most popular choice. To optimize Linux systems for real-time audio and low latencies there are two main strategies:

1. Using a PREEMPT_RT [4] kernel or a low-latency kernel with *rtirq*, assigning the audio processes higher priority [5] in the scheduling policy, reserving a processor core for the audio processes; all the usual Linux audio optimizations. Patchbox OS by Blokas [6] and Satellite CCRMA [5] are good examples for embedded Linux systems preconfigured that way.

2. The more sophisticated solution achieving supreme results at the cost of less portability involves using a Xenomai kernel. One design option is to remove the Audio IO from the kernels tasks completely to give it higher priority in the user space above the kernel. This is how the Bela [6] features outstandingly low latencies sitting on top of the Beaglebone Black. Elk Audio OS uses the Xenomai 4 dual kernel architecture [7] with a general-purpose kernel and a dedicated real-time kernel. Elk Audio [7] is designed to run on a Raspberry Pi system. Bela and Elk Audio both depend on custom hardware with audio codecs and additional features which attaches to their respective computing platform.

A comparison between the Xenomai based Elk Audio and a PREEMPT_RT system running on the same hardware (Raspberry Pi) can be found in Vignati et al. [8]. A third option would be to use a Linux OS capable processor, but running the process "bare metal", treating the processor like a microcontroller without the overhead of an operating system. This allows for an impressive embedded audio performance. An example of this technique on the Raspberry Pi is hinted to in Michon et al. [3]. The drawback is that it makes development more complex if there is need for OS features. Additional interfaces, graphics display, and everything that the OS would be already providing would need to be implemented once again.

[1] Intel acquired and absorbed Altera in 2015 re-branding the Altera products to the Intel trademark.

[2] *System-on-Chip:* an integration of core components of a computer in one chip (integrated circuit).

[3] https://github.com/chairaudio/SMC22-FPGA-accelerated-PD

[4] *System on Module*: a compact circuit board, typical for embedded systems. Usually populated with a processor or microcontroller and memory. It is connected to a host board where peripherals can be attached.

[5] By setting the "nice value".

[6] https://blokas.io/patchbox-os

[7] https://evlproject.org

A third category—besides fast micro controllers and low-latency OSs—leverages the computing power of FPGAs. Pfeifle and Bader [9] show, that FPGAs are particularly strong in computing physical models in the form of finite difference schemes, because these algorithms lend themselves to parallelization and their need for local storage variables is modest. For the usage in musical instruments FPGAs have two major drawbacks: designs need long development times and usually solve domain specific problems. E.g.: a design that models a vibrating membrane is inefficient in computing a string. Verstraelen et al. [10] summarize this problem as follows:

> Our conclusion is that FPGA technology has the ability to implement computational intensive real-time physical models of musical instruments, but the problem is to make this technology sufficiently flexible.

The usage of High Level Synthesis (HLS) languages as in Vannoy et al. [11] addresses the problem of long development times by using abstract behavioral models that automatically translate to complex digital logic designs and hiding away implementation details to the developer. Risset et al. [12] refine HLS by automatically translating FAUST programs—a data flow language specifically written for DSP programming—to FPGA digital logic. This is a very promising approach to speed up the development process in the context of musical applications and it would allow a broad library of open source code to use the massively parallel computing power on FPGAs with very low latencies.

But it does not ease one of the drawbacks of fixed digital logic architectures: It would be desirable to program FPGAs as flexible as software running on a CPU: algorithms or effect chains running on digital logic should be dynamically changed and be re-configurable by the user. I.e., it is not possible to load libraries of compiled code on an FPGA—while the system is running—to change the behavior of the circuit in a way that the designer has not foreseen. [8]

Verstraelen et al. [13], [10] approach this problem by designing a DSP processor that is tailored to compute a wide variety of physical models efficiently, without the need to change the architecture of the digital logic. Their solution is the design of a specialized DSP processor that uses the advantages of FPGA's distributed and massively parallel computing capabilities. This processor can be instantiated on an FPGA or be commercially produced as an ASIC. [9] On top of this DSP processor runs software written in a custom declarative language (conceptually a HLS language again) but comes with the benefits of faster compilation and re-programmability, similar to software on general purpose CPUs or DSPs. The application running on this DSP could theoretically be designed to be re-programmed in real-time, changing its data flow while audio is streaming through it, by inserting parts of pre-compiled code.

1.2 Motivation for this Research

Our motivation to investigate beyond the obvious choices of embedded platforms mentioned earlier was to create a unique platform for our instrument [14] enabling real-time acoustic excitation of complex physical models, possibly implemented in lumped mass-spring networks. This led to our wish for a powerful system which can compete with or even outperform desktop performance in key aspects. At the minimum it should offer *some* technological feature which is unique to the platform in comparison to the rest of the embedded system landscape. We briefly investigated parallel processing systems and therefore acquired an Nvidia Jetson Nano with the hope to implement physical models in shaders as discussed in Zappi et al. [15]. At the same time we also began to investigate the ARM-FPGA-SoC route and finally decided to focus on the latter due to the overall flexibility of the system, such as being able to integrate several audio input and output streams in digital logic design.

FPGAs are extremely powerful devices which can be configured by the user. This allows the FPGA to instantiate logic circuits like processors [10] which are not a mere emulation but truly identical to the original hardware down to the logic blocks. For this reason our particular development board is very popular in the retro-gaming scene. Gamers can experience arcane gaming platforms with the exact same performance and glitches like the original hardware. FPGAs are available with different amounts of logic cells and operators and can cost anywhere from 90 ct to 150.000 €. [11] Common uses for FPGAs are neural networks, simulations, database queries, LED-matrix controllers, prototyping of integrated circuits in hardware development, and cryptology. For DSP-applications a sufficient amount of available multiplier cells in the FPGA is essential. FPGAs can replace DSP chips and are also used in audio interfaces with hardware effects. Complex audio computations can be calculated massively parallel in FPGA logic with very little processing latency.

Since the Intel Cyclone V SoC combines FPGA and ARM processor in one chip, we can use the ARM CPU to run Pd. Pd is a graphical real-time programming environment. Using Pd in the product was a design decision to allow us to draft and deploy in the same language. This is liberating us from the complication of translating the entire algorithm from Pd to FPGA logic. Instead we can make better use of the FPGA focusing on specialized tasks which would be computationally expensive to implement on the CPU. Such computations can be a mass-spring-model, a large reverberation effect or similar DSP blocks which are then represented as an atom in the graph of a Pd patch. An additional benefit of using Pd is that the system can be programmed in real-time: that allows for fast prototyping and "tuning algorithms by ear", while the system is running.

[8] It can be argued that a part of the flexibility of software can be mimicked by a technique called *partial reconfiguration*, where parts of a logic design can be replaced. But—unlike in software—this method is limited by the hardware resources that the predefined reconfigurable area provides.

[9] *Application Specific Integrated Circuit*. ASICs are developed to solve domain specific problems and have medium to high volume productions.

[10] Processors implemented in FPGA hardware are called "soft cores".

[11] Based on a quick search on the website of a supplier. Many chips were not even available, probably due to the ongoing shortage.

We see "tinkerability" in a potential product as a great feature. In the musical instruments market it is not uncommon that the environment is open to the user so it can be tailored to specific needs or features contributed by the community. Products like the Organelle or the multi-effect devices from MOD-Devices are illustrating this paradigm. Exchanging parts of a DSP graph between the clocked FPGA and scheduler based Linux does however introduce complexity and comes with challenges for the timing of the processes. This paper is written to address these challenges and evaluate the proposed solutions.

Miller Puckette announced that he intends to port Pd to FreeRTOS so that it can run on the Espressif LyraT (ESP32) board. [12] Running Pd on FreeRTOS may further facilitate the FPGA integration and remove most of the obstacles caused by the different computing approaches. We look forward to test this when it becomes available.

The motivation for this research was to answer the following questions: "Is it possible to include an FPGA coprocessor in the signal processing chain while using a proven audio programming environment (Pd) on a Linux OS? Will the performance of such system be sufficient for physical modeling synthesis and the latency acceptable?"

2. HARDWARE AND SOFTWARE COMPONENTS

We chose the DE10-Nano as a platform because it's one of the entry products which nevertheless features a reasonably sized FPGA along with an ARM core for running Linux. The physical modeling algorithms we intend to run, require sufficient hardware multipliers and on-chip ram. The DE10-Nano features 112 fixed point 27 Bit multipliers, which can run at a clock rate of 350 MHz. The ARM is a dual core running at 800 MHz clock rate each.

The DE10-Nano board does not have an audio codec, therefore we had to install it separately. We decided for a Mikroe-506 board featuring a Wolfson WM8731 codec. Another component that is missing on the SoC is the I2S interface to stream audio data. We build upon work of Bjarne Steinsbø for Terasic's DE1-SoC who wrote both—the I2S hardware interface and the accompanying opencores-I2S device driver. [13] This gives us Verilog code to instantiate the necessary hardware I2S interface inside the FPGA fabric (see Fig. 1).

For the codec chip itself we configured the Altera Linux Kernel 5.4.54-LTS to include the Wolfson 8731 codec driver and added support for the ALSA [14] System on Chip (ASoC) simple card. [15] The ASoC simple card driver glues together the opencores-I2S device with the WM8731 codec and is the interface between the OS (ALSA) and these components (See Fig. 2).

Finally we installed Pd v.0.51.2 with ALSA support on the provided Ångström v2016.12 Linux console image. The custom digital logic for the sound card is completely opaque to user space software. We can simply use the standard ASoC simple card as an input and output device. This allows for a standard user experience in Pure Data (or any other audio software on the platform).

On the other hand, the DSP subsystem that is instantiated in FPGA logic needs a non standard approach for communication and audio data exchange. To address this issue, we wrote a Pd external that sends and receives audio blocks and communication data with the FPGA (see section 3.2).

3. DATA FLOW BETWEEN PURE DATA AND FPGA

On this SoC platform, FPGA and ARM share the same die—this is beneficial for fast data transfers at low latency. The Cyclone V SoC features an industry standard AXI bridge to allow high speed communication between both devices.

3.1 Memory Mapping

The AXI bridge is at a hardware address that is normally only visible to the kernel. To access this region from user space, we map the hardware address from `/dev/mem` to our local program memory using `mmap()`. This sends a request to the kernel to set up page tables so we can redirect to hardware addresses. For this we need read/write privileges to `/dev/mem`. On the FPGA side we can define memory interfaces and other peripherals to be mapped to sub-addresses of the AXI bridge. To read and write data to the bus, we can then just map the hardware (sub-) addresses from the AXI bridge to our local program memory and access it through a pointer.

In our case, the physical memory address of the Heavy Weight AXI bridge is given to be `0xC000_0000`. On the FPGA side we instantiated several on-chip memory interfaces as shown in Fig. 3. The AXI bridge sub-addresses are automatically offset to local addresses in the FPGA memory space, so we can directly read and write to them in a C program as if they where local arrays.

3.2 Pure Data External

Pd is a real-time data flow programming environment. It uses graphical objects ("atoms") which typically have sinks ("inlets") and sources ("outlets") that allow for interconnections to route data or signals in the graph. Pd ships with a number of core objects that solve standard problems. The user can extend these objects by "abstractions", which are itself written in Pd, similar to writing a reusable function in other programming languages. Objects written in C and compiled for Pd are called "externals". Libraries of abstractions and externals mostly maintained by third parties can be distributed via Deken; Pd's integrated package manager.

Enabling the FPGA to exist as a node of the DSP graph of Pd, we wrote an external which takes care of copying the data from the CPU to the FPGA and back.

To allow audio signals and control data to be sent from Pd to the FPGA on-chip memory we use the memory mapping technique described above. Blocks of audio samples that arrive at the externals input can be directly written to the

[12] PD-dev Mailinglist 2021-07-13 `https://lists.puredata.info/pipermail/pd-dev/2021-07/022773.html`

[13] The source code is available at `https://github.com/bsteinsbo/DE1-SoC-Sound`.

[14] *Advanced Linux Sound Architecture*

[15] Available at `https://github.com/altera-opensource/linux-socfpga`.

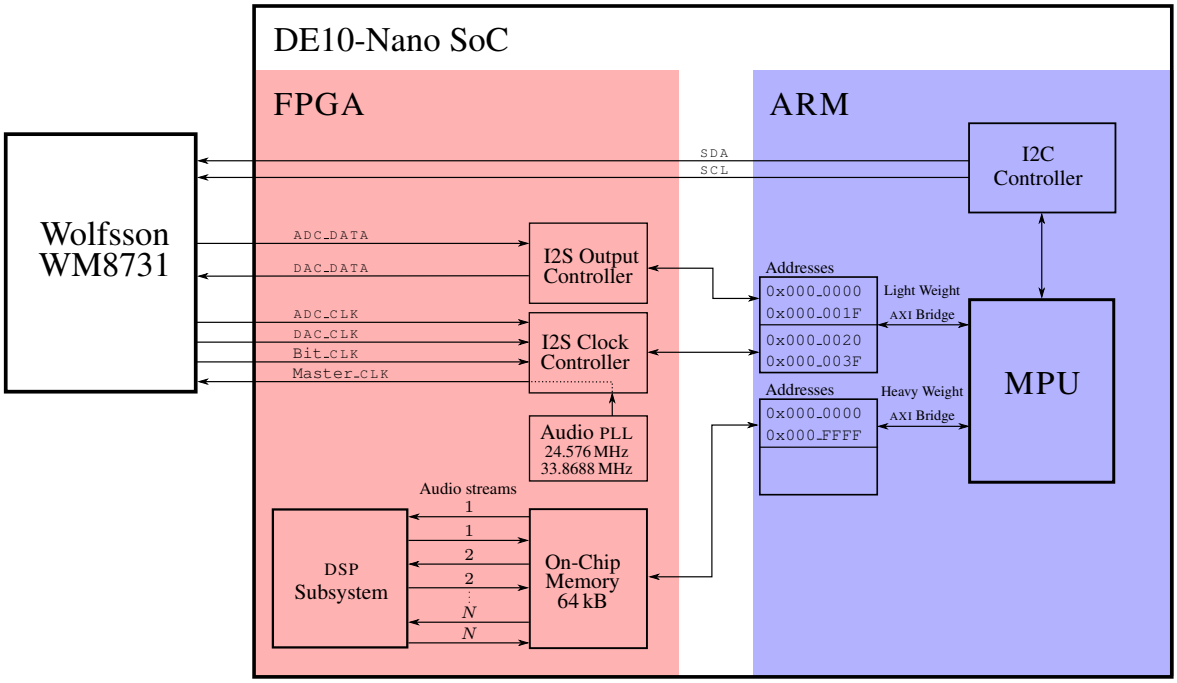

Figure 1. Hardware Components.

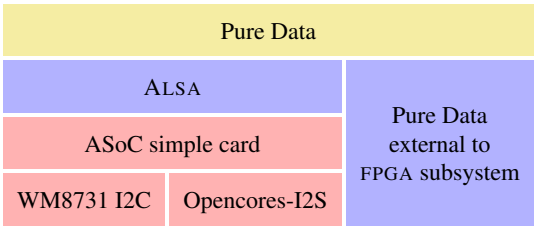

Figure 2. User Space, Kernel Space, Kernel and User Space

memory mapped arrays and will then be sent over the AXI bridge to FPGA on-chip memory.

In the external's constructor we need to set up the memory map and retrieve a pointer to our hardware for later use as shown in listing 1. In the DSP routine of the external (the DSP callback) we can then use the pointer to read and write samples to this buffer as shown in listing 2. Note that we can detect dropouts, by reading and clearing a flag that the digital logic on the FPGA sets when it is finished processing the samples. We can also tell the FPGA how many samples it should process by writing this information to the on-chip memory. This way, we can change our block size in the Pd patch and the FPGA will immediately follow the change, shrinking or increasing the amounts of samples to process dynamically.

```
// opening /dev/mem for mapping to local mem.
x->fd = open("/dev/mem", (O_RDWR | O_SYNC));
// mapping input sample buffer
x->in_smp_buf_map = mmap(NULL,
 IN_SMP_BUF_SPAN, //size of map
 ( PROT_READ | PROT_WRITE ),
 MAP_SHARED, //synchronize data
 x->fd, 0xC0004000); //hardware address
x->in_smp_buf_ptr=(int *)(x->in_smp_buf_map);
// mapping output sample buffer
x->out_smp_buf_map = mmap( NULL,
 OUT_SMP_BUF_SPAN,
 ( PROT_READ | PROT_WRITE ),
 MAP_SHARED,
 x->fd, 0xC0005000); //hardware address
x->out_smp_buf_ptr=(int *)(x->out_smp_buf_map);
```

Listing 1. Mapping on-chip memory to the local sample buffers.

```
...
// data struct for external
t_fpga_tilde *x = (t_fpga_tilde *)(w[1]);
// input samples to external
t_sample *audio_in=(t_sample *)(w[2]);
// output samples from external
t_sample *audio_out=(t_sample *)(w[3]);
// current block size
int n = (int)(w[4]);

if (x->out_smp_buf_ptr[0] != 0xabcd) {
   // fpga not finished
   pd_error(x,
     "FPGA buffer dropout detected");
 }
// copy data to fpga
memcpy((void*)&x->in_smp_buf_ptr[512],
  (const void*)audio_in,
  4*n);
// get old (processed) data back from FPGA
memcpy((void*)audio_in,
  (const void*)&x->out_smp_buf_ptr[512],
  4*n);

// tell FPGA number of samples to process
x->in_smp_buf_ptr[1] = n;
// pure data: buffer ready flag
x->in_smp_buf_ptr[0] = 0xabcd;
// clear FPGA ready flag
```

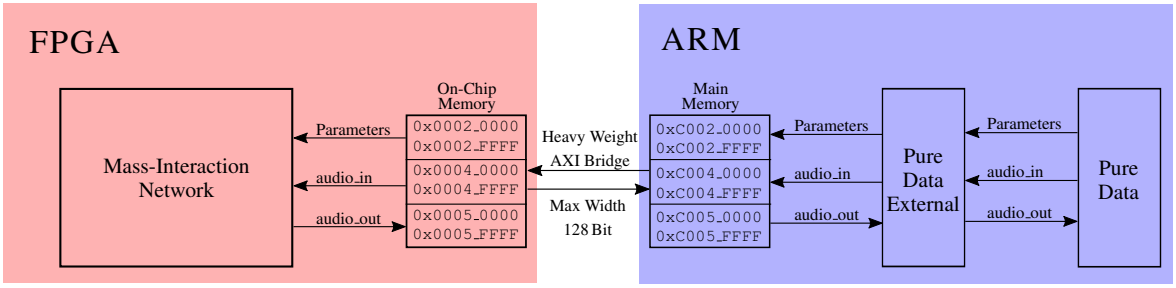

Figure 3. Data flow between Pd and FPGA.

```
x->out_smp_buf_ptr[0] = 0x0;
...
```

Listing 2. DSP callback function of the Pd external. Using flags, FPGA dropouts can be detected.

3.3 FPGA Digital Logic

On the FPGA side, digital logic was written to read out the on-chip memory modules and feed the data forward to other DSP processes.

Index	ARM Data	FPGA Data
0	ARM Ready	FPGA Ready
1	Block Size	Buffer Interval Count
2-511	Reserved	Reserved
512-1023	Samples	Samples

Table 1. Data structure of the buffer exchanged with the FPGA. Data size is 32 bit.

For measuring jitter and system latency, we implemented a state machine that waits for new data, and then simply copies the audio input to the audio output buffer and sets a flag when it is done. An additional counter to measure the time interval between two buffer arrivals was also implemented. Table 1 shows the structure of exchanged buffers. The ARM and FPGA buffers each reside in a dedicated on chip RAM as depicted in Fig. 3.

Our real-world DSP application is a mass-spring model, which unfortunately will have to be described in another paper since we hit the 8 pages limit here.

3.4 Buffer Transfer Speed

Fig. 3 shows how data flows between Pd and the FPGA portion of the SoC: Audio samples, parameter- and control data is sent through the *Heavy Weight AXI Bridge* which is the fastest hardware interface available on the ARM SoC. This bus is 128 Bit wide and runs at a maximum speed of 200 MHz. This leads to a theoretical maximum bandwidth [16] of 3.2 GB/s between FPGA and ARM. [16]. However, the achievable speed varies with the applied memory access techniques, the OS and the transferred data sizes.

[16] The bridge is 128 Bit wide, with a maximum clock speed of 200 MHz.

On Ångström embedded Linux a buffer of 64 32-bit integers transfers happen to be similar for `memcpy()` transfers and DMA transfers—they are between 60-90 MB/s fast [16]. For bigger buffer sizes, using DMAs is significantly faster: A 256 samples buffer has a transfer rate of 250 MB/s using DMA and 60 MB/s using `memcpy()`. Top speeds for DMA transfers are at 8192 samples with 600 MB/s throughput while `memcpy()` peaks out at 60 MB/s. Xilinx Zynq platforms have similar transfer speeds in our system configuration (see [17]).

Bruce R. Land reports read/write rates of 28 MB/s using single element access (no `memcpy()`) and 266 MBytes/s using DMA for a buffer size of 10.000 32-bit integers. [17]

4. CONFIGURING PURE DATA FOR REAL-TIME BUFFER TRANSACTIONS TO FPGA HARDWARE

A particular challenge in real-time systems is that information needs to be passed just in time to form a continuous data flow. We need to make sure that buffers are passed from Pd to the digital logic design at regular intervals. When intervals vary, the amount of time that can be spend for computational tasks in the DSP subsystem will vary too.

However, operating systems are usually optimized to solve concurrent tasks and prioritize those that the scheduler assumes to be critical. Running programs may signify the scheduler that they finished their task by *sleeping*. After either a predefined interval or an external wake up mechanism they will be called to work again, but must compete with other tasks for computational time. This leads to the situation that data is typically processed in irregular intervals, based on the arbitration of the scheduler.

Pd was written to run on non-real-time systems and is capable to deal with task scheduling by the OS—essentially allowing a balancing of computational load between its audio tasks and parallel tasks of the OS.

Pd handles this load balancing with its own internal scheduler: By default it computes the message and audio flow for blocks of 64 samples each [18, p. 63]. When the computation of one block of message and audio data flow is completed, it *sleeps*, which allows the OS's scheduler to delegate the free compute time to other processes. The

[17] `https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/FPGA_addr_index.html`

sleep interval in Pd's scheduler is calculated to be of "reasonable" size (Pd's global delay setting divided by four). In our case this was 0.75 ms. [18] That means, when the last block of audio was processed and no more data is in the pipeline, the scheduler sleeps [19] for 0.75 ms and frees up the processor. During that time a couple of new blocks (here, a block of 16 samples had a real-time duration of 0.33 ms) accumulate in the processing pipeline and will get processed in a fast burst when Pd awakes from sleep.

This burst processing behavior is undesirable in the presence of external processes that need to be synchronized to each sample block. When a burst occurs, the external process (in our case the digital logic on the FPGA) has very little time to compute the sample block before it must be returned to the main process. Fortunately Pd provides means to reduce sleeping time and thus get more predictable block process intervals. By adjusting the *sleepgrain* parameter—either through a startup flag or a Pure Data external—the timing of block calculations can be altered. In short: the smaller the *sleepgrain* size, the less jitter will occur on the block processing intervals but the more function calls and context shifts between the OS and Pd will occur that will slow down other processes running on the OS.

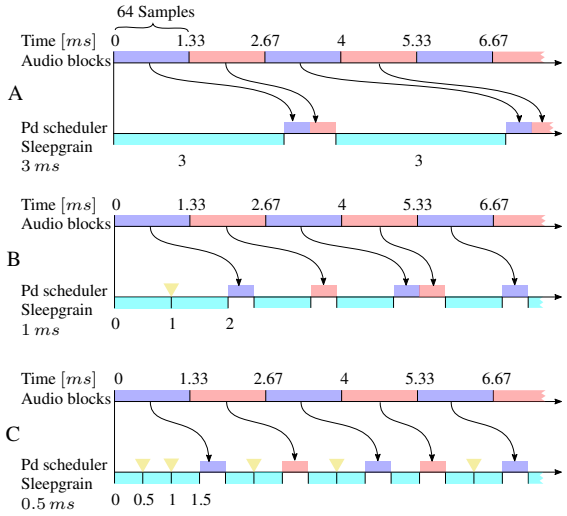

Figure 4. Processing of audio blocks with the Pd scheduler set to different *sleepgrain* sizes. A: Large *sleepgrain* sizes results in audio blocks being processed in bursts. B: A *sleepgrain* size with a similar duration like the blocksize in time results in few overhead function calls but causes jitter. C: Smaller *sleepgrain* sizes generate many overhead function calls but delivers a steadier computation of audio blocks. Overhead function calls, Sleep.

Fig. 4 illustrates how the *sleepgrain* parameter generates an overhead in function calls for small sizes. Note that this overhead does not harm the audio process, since the sleep calls are only generated cyanonce all audio and messaging work is done. The sleep function calls show a significant loading effect, when Pd is mostly idle, but if we run an audio process with more computational load to begin with (as shown in Fig. 5) this parameter has less influence on the total computational load. We only see a slight increase for small *sleepgrain* sizes, since Pd spends less time in idle state where additional function calls are generated. Even though DSP load increases for smaller *sleepgrain* sizes, buffer dropouts decrease. A side effect of this can be a sluggish or unresponsive graphical user interface, because background tasks will be slowed down.

To reduce overall system latency and allow the transfer of smaller audio blocks without bursting, we compiled Pure Data with an audio block size of 16 samples. This comes with the trade-off of processing overhead through function calls.

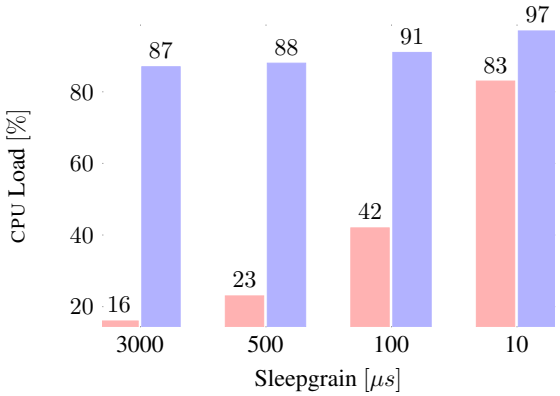

Figure 5. Low DSP load, High DSP load, The shorter the sleep time in Pd, the more virtual load will be on the CPU. The CPU load was measured with the in-built meter in Pd, averaged over 10 seconds.

5. JITTER/LATENCY MEASUREMENT SCHEMES

The total input to output latency measurements were taken using a square wave generator and an oscilloscope. Using two oscilloscope channels, the time difference between the original signal and the one sent through the device was measured. The period of the square wave was chosen to be longer than the latency of the system.

To quantify the block interval jitter we instantiated a counter in digital logic. This counter was driven by a low jitter 50 MHz clock source from external hardware. Each time a block of audio was written to the digital logic, the counter was reset and its value reported to a Pd external. This way we can get reliable timing information without needing to use system calls in our audio process functions.

Finally, we varied the block and *sleepgrain* sizes and calculated the block interval time from the counter value.

We took the minimum of the block interval time of 10,000 measurements for each setting and used this smallest interval to estimate the maximum use of capacity for the FPGA as follows:

[18] We found the sleepgrain size by calling `get_sys_sleepgrain()` in the sleepgrain external by IOhannes m zmölnig `https://git.iem.at/pd/zexy`. This agrees with the global delay setting being 3 ms.

[19] See `http://msp.ucsd.edu/Pd_documentation/x3.htm#s4`.

$$\text{FPGA}_{\max} = \frac{\min\{I_1, I_2, \ldots, I_m\}}{I_{rt}} \cdot 100 \qquad (1)$$

where:

$\text{FPGA}_{\max}$ = maximum use of FPGA capacity [%]
I_m = measured block intervals, $m = 10000$
I_{rt} = block interval in real time.

This number tells us the maximum load time of the FPGA where the likelihood of dropouts is very low.

5.1 Results

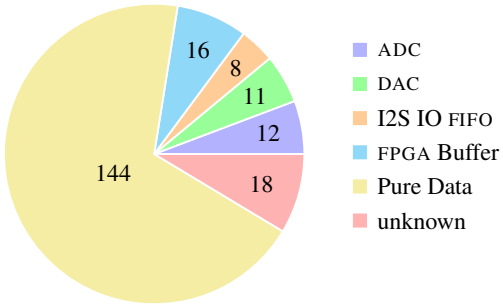

Figure 6. System latencies in samples. Total is 209 samples.

The total measured roundtrip latency was 4.36 ms. This accounts for all delays in the system: ADC and DAC conversions (23 samples), I2S sample buffer (8 samples), Pure Data's "Delay" setting (3 ms) and FPGA buffering (16 samples). This delay accounts for 209 samples at a sample rate of 48 kHz. Jitter was too small to measure with our measurement techniques and was lower than 0.1 ms. We did not investigate where the unknown additional latency of 18 samples arises from.

The share of delay added by the FPGA data transmission was 0.33 ms or 16 samples at 48 kHz.

Since we are using Pd on an embedded system where it is the process of highest priority, we can reduce the sleepgrain size to even 10 μs without major drawbacks. This system exhibits reasonable low latency for use as a digital musical instrument (4.36 ms) and leverages up to 88% of the theoretical processing power of its FPGA coprocessor (see Fig. 7).

6. CONCLUSIONS

We successfully demonstrated how to offload some of the complex audio processing to an FPGA coprocessor from an ARM processor running a Linux OS. From within a Pure Data application the FPGA computing is handled by an external. The latency is satisfactory, the system is stable (yet there is still room for further optimizations). A brief video can be found at `https://vimeo.com/668575690`.

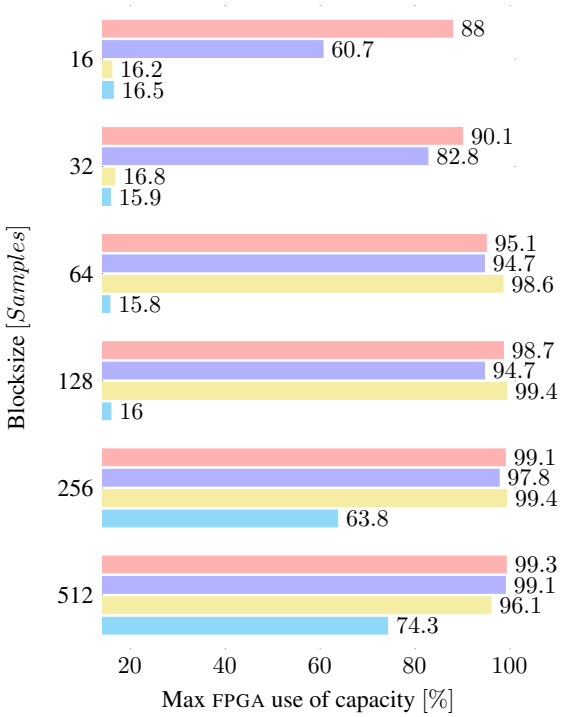

Figure 7. Sleepgrain size 10 μs, 100 μs, 500 μs, 3000 μs. The shorter the sleep time and the bigger the block size in Pd, the more compute time can be spent on the FPGA.

7. FUTURE WORK

- Allow multiple FPGA Pd-externals to coexist in one patch.
- Allow for uploading new bitstreams (the patterns for the cell fabric layout on the FPGA defining the DSP graph) to the FPGA from within Pd.
- In extension of the previous point: Let the user edit Verilog/VHDL from within Pd, compile on the fly and upload to FPGA. This would require open toolchains which are not available yet.
- Extend the simple mass-string models to more complex mass-interaction networks with non-linear interactions like collisions, penetrations, attraction, gravitation, parameters which are dynamic and/or conditional, etc.
- To further reduce latencies alternative hard- and software designs should be evaluated, such as replacing the Altera Linux kernel with a Xenomai 4 dual kernel or implementing the techniques found in BeagleRT [6].
- Speeding up Pd processing by using heavy compiler [20]. A benchmark must quantify the performance gain.
- Using the Faust to Verilog/VHDL compiling toolchain as in Risset et al. [12].

8. REFERENCES

[1] A. P. McPherson, R. H. Jack, and G. Moro, "Action-Sound Latency: Are Our Tools Fast Enough?" in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Brisbane, Jul. 2016, p. 6.

[2] J.-F. Charles, C. C. Solares, C. T. Tobon, and A. Willette, "Using the Axoloti Embedded Sound Processing Platform to Foster Experimentation and Creativity," in *Proceedings of NIME 2018*, Blacksburg, 2018, p. 2.

[3] R. Michon, Y. Orlarey, S. Letz, and D. Fober, "Real Time Audio Digital Signal Processing With Faust and the Teensy," in *Proceedings of the 16th Sound & Music Computing Conference*, Malaga, May 2019, p. 7.

[4] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_rt," *ACM Computing Surveys*, vol. 52, pp. 1–36, Jan. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3297714

[5] E. Berdahl and W. Ju, "Satellite CCRMA: A Musical Interaction and Sound Synthesis Platform." in *NIME*, 2011, pp. 173–178.

[6] A. McPherson and V. Zappi, "An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black," in *Audio Engineering Society Convention 138*, Warsaw, May 2015. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=17755

[7] L. Turchet and C. Fischione, "Elk Audio OS: An Open Source Operating System for the Internet of Musical Things," *ACM Transactions on Internet of Things*, vol. 2, p. 18, Mar. 2021.

[8] L. Vignati, S. Zambon, and L. Turchet, "A Comparison of Real-Time Linux-Based Architectures for Embedded Musical Applications," *Journal of the Audio Engineering Society*, vol. 70, pp. 83–93, Jan. 2022, publisher: Audio Engineering Society. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=21553

[9] F. Pfeifle and R. Bader, "Real-time Finite Difference Physical Models of Musical Instruments on a Field Programmable Gate Array (FPGA)," in *Proc. of the 15th Int. Conference on Digital Audio Effects*, York, UK, 2012, p. 8.

[10] M. Verstraelen, F. Pfeifle, and R. Bader, "Feasibility analysis of real-time physical modeling using WaveCore processor technology on FPGA," in *Proceedings of the Third Vienna Talk on Music Acoustics*, University of Music and Performing Arts Vienna, Sep. 2015.

[11] T. Vannoy, T. B. Davis, C. Dack, D. Sobrero, and R. K. Snider, "An Open Audio Processing Platform using SoC FPGAs and Model-Based Development," in *New York*, 2019, p. 8.

[12] T. Risset, R. Michon, Y. Orlarey, S. Letz, G. Müller, and A. Gbadamosi, "Faust2FPGA for Ultra-Low Audio Latency: Preliminary work in the Syfala project," in *Proceedings of the 2nd International Faust Conference*, Paris, 2020, p. 10. [Online]. Available: https://ccrma.stanford.edu/~rmichon/publications/doc/ifc-20-faust2fpga.pdf

[13] M. Verstraelen, J. Kuper, and G. J. M. Smit, "Declaratively programmable Ultra Low-Latency Audio Effects Processing on FPGA," in *Proc. of the 17th Int. Conference on Digital Audio Effects*, Erlangen, Sep. 2014, p. 8.

[14] M. Neupert and C. Wegener, "Interacting with digital resonators by acoustic excitation," in *Proceedings of the 16th Sound & Music Computing Conference*. Malaga: Universidad de Málaga, May 2019, pp. 80–81. [Online]. Available: http://smc2019.uma.es/articles/D1/D1_01_SMC2019_paper.pdf

[15] V. Zappi, A. Allen, and S. Fels, "Shader-based Physical Modelling for the Design of Massive Digital Musical Instruments," in *Proceedings of the 2017 International Conference on New Interfaces for Musical Expression*, Aalborg, May 2017, p. 6.

[16] R. F. Molanes, J. J. Rodriguez-Andina, and J. Farina, "Performance Characterization and Design Guidelines for Efficient Processor–FPGA Communication in Cyclone V FPSoCs," *IEEE Transactions on Industrial Electronics*, vol. 65, pp. 4368–4377, May 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8082548/

[17] R. F. Molanes, L. Costas, J. J. Rodriguez-Andina, and J. Farina, "Comparative Analysis of Processor-FPGA Communication Performance in Low-Cost FPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 17, pp. 3826–3835, Jun. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9165225/

[18] M. Puckette, *The theory and technique of electronic music*. Hackensack, NJ: World Scientific Publishing Co, 2007, oCLC: ocn166265250.

[19] J. Leonard, J. Villeneuve, R. Michon, Y. Orlarey, S. Letz, A. Three, and A. Four, "Formalizing Mass-Interaction Physical Modeling in Faust," *Stanford University*, p. 7, 2019.

[20] G. Moro, A. Bin, R. H. Jack, C. Heinrichs, and A. P. McPherson, "Making High-Performance Embedded Instruments with Bela and Pure Data," in *Proceedings of ICLI*, University of Sussex, Brighton, UK, Jun. 2016, p. 5.