Real-Time Audio Processing on the T-CREST Multicore Platform

Daniel Sanz Ausin, Luca Pezzarossa, and Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby
Email: s142290@student.dtu.dk, [lpez, masca]@dtu.dk

***Abstract*—Multicore platforms are nowadays widely used for audio processing applications, due to the improvement of computational power that they provide. However, some of these systems are not optimized for temporally constrained environments, which often leads to an undesired increase in the latency of the audio signal. This paper presents a real-time multicore audio processing system based on the T-CREST platform. T-CREST is a time-predictable multicore processor for real-time embedded systems. Multiple audio effect tasks have been implemented, which can be connected together in different configurations forming sequential and parallel effect chains, and using a network-on-chip for intercommunication between processors. The evaluation of the system shows that real-time processing of multiple effect configurations is possible, and that the estimation and control of latency ensures real-time behavior.**

I. Introduction

Multiprocessor systems are frequently found in the field of audio signal processing. Some examples are audio software environments that run on multicore processor computers, or embedded audio multicore platforms that are found in many applications, such as hearing aids or portable mobile devices [1].

It is frequent in audio signal processing systems that multiple types of processing are applied to the signal. As an example, guitarists use many different audio effects to process the guitar signal before it arrives at the loudspeaker. These effects could be distortion effects, filters, delays, and so on. Usually, the effects are connected in a sequential way, where the output signal of one effect is the input of the next one in the chain. But it might also be the case that the signal is split into parallel chains to be processed separately, and then merged together, forming a more complex graph. An example of this is shown in Figure 1.

In this work, we focus on real-time audio processing applications, which means that the processing must be applied within an interval of time that ensures that the input-to-output latency of the signal is imperceptible for the human ear. The temporal behavior of the processing system must be completely predictable in order to provide real-time guarantees. To achieve this, the elements of the system, such as the buffers of each effect that processes the signal, need to be designed in a way that the input-to-output latency is kept under a certain limit.

This paper presents a real-time audio processing system targeted to the time-predictable T-CREST multicore platform. T-CREST is a network-on-chip (NoC) based platform for hard real-time applications. In this work, typical digital signal processing (DSP) algorithms for audio applications have been implemented. The effects can then be connected together in the same way as shown in Figure 1 in the multicore platform. The effects are statically mapped to different cores, and the NoC is used as the main communication resource between processor cores. The three main contributions of this paper are:

- The design and implementation of a flexible and scalable real-time audio processing system, supporting different setups and combinations of audio effects in the multicore platform;
- The real-time software implementation of common audio effects, with strong focus in worst-case execution time (WCET) predictability and reduction;
- A fully functional open-source real-time application, serving as a use case for the T-CREST platform and, potentially, for other real-time multicore architectures.

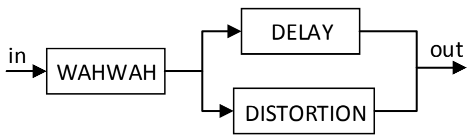

Fig. 1. Example of sequential and parallel chains of audio effects.

The architecture presented in this paper represents an alternative approach to conventional audio processing systems, where the latency is a result of fixed buffer sizes and not a design parameter, which could lead to the loss of real-time behavior [2]. In our approach, the effects are statically allocated on the cores of the platform, and the parameters of the resources (processors, communication channels, buffers, and so on) are set at compile-time in order to have full control and predictability of the signal latency, thus guaranteeing real-time audio processing.

Further details of the work presented here can be found in the master thesis [3]. All the code is available in open source. Additionally, the platform uses an audio interface and a software API to access audio data in real time [4].

This paper is organized in seven sections: Section II presents related work on multicore real-time audio processing. Section III gives a brief background on topics related to audio signal processing, and presents a general overview of the T-CREST multicore platform. Section IV describes the individual effects designed for the Patmos processor, and Section V explains how multiple effects are combined together in the multicore platform. Section VI evaluates the proposed approach in terms of its real-time behavior. Finally, Section VII concludes the paper.

II. Related Work

As it is explained in [5], when multicore platforms are used for real-time audio signal processing, a higher computational power is achieved. An audio processing system is a collection of tasks that communicate between each other. These tasks can execute on multicore platforms. Our system uses multiple cores for parallel tasks as well.

In multicore platforms, an optimal distribution of the tasks or threads among the computational elements of the system is essential to utilize the resources effectively. The work presented in [6] presents two scalable approaches for the optimization of thread distribution in multicore platforms in object-based real-time audio processing environments. Here, a Signal Flow Chart is defined as a graph containing all the components that process the audio signal. In our system, similar flow charts are defined, and we refer to the processing components as effects.

Our work uses a similar approach to the flow decomposition algorithm presented in [7], where a sequence of tasks is executed concurrently by chaining their inputs and outputs through buffers, forming sequential and parallel chains where the audio signal is split/merged in forks/joins. The result is a system with high computational parallelism due to pipelining of the audio stream through the components.

Combining different types of processing cores into the same platform might be an efficient solution to cover a wide range of processing algorithms. It is common to find desktop computers, laptops or other devices nowadays with multiple cores for different purposes. As it is shown in [8] and [9], Graphics Processing Units (GPU) are very widely used nowadays for audio processing, and can reduce execution time considerably due to their high parallelism in data processing. The work presented in [10] focuses on the acceleration of real-time audio DSP algorithms (IIR/FIR filters, modulation algorithms, and so on) by combining the usage of CPUs and GPUs in a multicore system. This work also states that transfers between processors are usually an important bottleneck, and therefore supports the need of an efficient intercommunication system. Although the T-CREST platform used in this project consists of homogeneous processors, the system can easily allow the integration of different cores in the network.

The work presented in [5] explains the undesired effect that buffering has in latency, and suggests limiting or avoiding it when possible. In addition, the work presented in [2] explains the importance of latency control and estimation in real-time audio systems, which is often not considered in software-based digital audio workstations used for live music performances. It also provides a tool to measure the latency of hardware components and software DSP algorithms. Prediction and reduction of the audio signal latency is one of the main design concerns of our work, as we are focusing on real-time audio processing.

III. Background

This section first classifies the audio effects used in this work. It then provides background information regarding real-time digital audio signal processing. After that, the concept of homogeneous synchronous data flow is presented, and finally a general overview of the T-CREST multicore platform is given.

A. Classification of Audio Effects

There are many different digital signal processing algorithms that are used in audio applications, which can range from enhancing some properties of the signal, to extracting certain information from it. Their main computational requirements are multiply-accumulate operations and memory access instructions. Some of these algorithms are used to implement audio effects for music applications [11]. The ones that have been used in this work are classified as follows:

- Filters: infinite impulse response (IIR) filters have been mainly used to implement equalization filters, such as low-pass, high-pass, band-pass and band-reject filters. Comb filters have been used for conventional delay or echo effects.
- Modulation effects: it is common to add temporal variations to some parameters of the audio signal or the effects. In the tremolo effect, the amplitude of the signal is modulated, while in the vibrato effect, the pitch of the signal is altered. More complex modulation effects include variations of filter parameters: in the wah-wah effect, the central frequency of a band-pass filter is modulated, while the chorus effect modifies the length of 2$^{\text{nd}}$ order comb filters.
- Non-linear processing effects: they are also known as wave-shaping effects, and are used to enhance the character of the signal by adding some harmonic content to it. The overdrive effect is very commonly used in music applications, and applies subtle wave-shaping to the signal. On the other hand, the effect known as distortion can be much stronger, as it generates a higher harmonic content.

B. Real-Time Audio Signal Processing

The most common operations required in audio signal processing are multiply-accumulate and memory load/store instructions, to access buffered audio samples, filter coefficients or other parameters [1].

An important property of a real-time digital audio processing system is that the processing time per sample must be smaller than the sampling period in order to avoid interruptions in the output signal. This is shown in Equation 1, where t_p is the processing time per sample and F_s is the audio sampling rate.

$$t_p \leq \frac{1}{F_s} \tag{1}$$

Another characteristic of real-time audio processing systems is that the latency of the signal from input to output must be kept

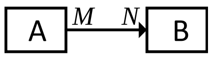

Fig. 2. Example of two actors with different data rates on a SDF model.

within an acceptable time interval. In music applications, the processing must be perceived as instantaneous by the human ear. Usually, latency values of just a few milliseconds are considered acceptable [2]. In this work, a latency limit of 15 ms is used as a reference: under this value, the human ear is incapable of noticing any delay.

C. Homogeneous Synchronous Data Flow

The implemented multicore audio processing system is an example of a homogeneous synchronous data flow (SDF) [12] application. SDF is a model of computation where a digital signal is processed by actors in a static order of execution. When an actor has enough input tokens, it fires (i.e., starts the computation) to produce output tokens. In homogeneous SDF, the actors have a static data rate, which is the amount of tokens they need in order to fire.

Audio processing is a multi-rate SDF model, where the firing rates of the actors in the system might not be identical. Figure 2 shows an example where actor A produces M tokens each time it fires, and actor B requires N input tokens to fire.

Balance equations need to be defined, to guarantee the correct communication and synchronization of all actors in the system. The balance equation for the simple example of Figure 2 is shown in Equation 2, where q_x indicates how many times actor X must fire in order to have a constant data rate in the overall system.

$$q_A M = q_B N \tag{2}$$

D. The T-CREST Multicore Platform

The T-CREST platform [13] is used for the implementation of the audio processing system. The goal of the T-CREST project is to develop a fully time-predictable multicore processor for embedded hard real-time applications. For this, many resources are available, the most important ones for this project being the Patmos processor, the Argo NoC, and the Platin time-analysis tool. All these components have been designed for WCET reduction and predictability. The version used in this work is a 4-core platform, as shown in Figure 3, that runs on an Altera DE2-115 FPGA board [14].

Patmos is a RISC-style time-predictable processor [15]. It is the main computational resource of the T-CREST platform, and has been used here for the computation of audio processing algorithms. Patmos has a set of local memories that allow faster data and instruction access and tighter WCET bounds. These are the data and instruction caches and scratchpad memories (SPM), which can be explicitly used by the programmer.

Patmos has I/O devices mapped to the local memory address space. Among others, an audio interface [4] is found here, which implements the communication between the processor and the WM8731 audio CODEC [16] found in the DE2-115 board.

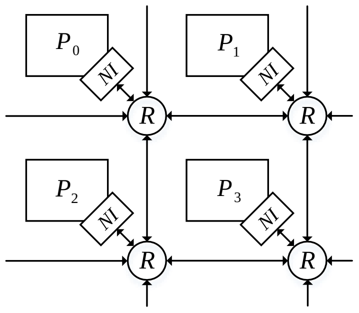

Fig. 3. Representation of the 4-core T-CREST platform. The Argo NoC is shown with its network interfaces (NI), routers (R) and links between routers.

Argo is a time-predictable statically-scheduled NoC [17], which implements the message-passing between the Patmos processors of the platform. Argo uses time-division multiplexing to share its hardware resources over time, allowing data packets to travel from source to destination within a guaranteed time interval, making Argo ideal to reduce and pre-estimate WCET. In the audio processing platform, the audio signal travels through the Argo NoC from core to core.

The Platin tool kit [18] has been used for WCET analysis of each one of the implemented audio effects. Platin uses a model of the Patmos processor and information of the compiled program to analyze the WCET bound of a given function. WCET analysis is essential for any real-time system: in audio processing, it is used to know if real-time processing is possible, and how many audio effects can be processed together in the platform (i.e., how much processing can be applied to the audio signal before any audio interruptions could happen).

IV. Design of Audio Effects for the Patmos processor

Multiple audio effects have been implemented in software, using an object-oriented style approach, where each audio effect is represented by a different class. This modular approach is common in audio signal processing, because it allows to easily instantiate effects and combine them with others. The parameters of each effect are stored as a data structure in the local SPM of Patmos. This ensures single-cycle access to parameters, essential for a short WCET, as it avoids relying on caches. In most effects, a block of the latest audio samples needs to be buffered. These samples can be stored in the SPM. However, in some cases, larger audio buffers need to be kept in the external SRAM memory.

Figure 4 shows the flow of the audio signal in the single-core T-CREST platform. The input and output buffers are part of the audio interface component, and each effect is processed in software inside the Patmos processor.

WCET analysis of each effect is essential to guarantee that all of them can be processed in real time. The chosen audio sampling rate is $F_s = 52.083\ kHz$, which is equivalent to a sampling period of 1536 clock cycles, for the 80 MHz processing clock of Patmos. Those 1536 cycles set the limit for real-time processing, thus it is important to guarantee that

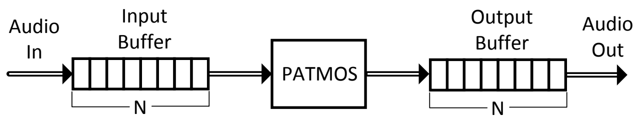

Fig. 4. Representation of the audio signal flow on the single core T-CREST platform, with input and output buffers of the same size, *N*.

all the effects can be computed in a shorter amount of cycles in all cases. The results of WCET analysis are shown in the evaluation in Section VI.

From the WCET analysis, an *utilization* value can be defined as shown in Equation 3, for a sampling period of 1536 cycles. The utilization of each effect gives an idea of how much the computational resources of the processor are being used.

$$U_{FX} = \frac{WCET_{FX}}{1536} \tag{3}$$

As an example, if an effect a has $U_a = 65\%$, it means that the processor will be computing during at most 65% of the time, and it will be idle at least for 35% of the time.

V. Audio Processing in the T-CREST Platform

This section presents the multicore audio processing platform and is divided in 3 subsections. At first, the static task allocation is explained; after that, some technical characteristics are presented, which explain how the effects are computed, synchronized and communicated; finally, the calculation of the latency of the signal is shown.

A. Static Task Allocation

Our system is designed in a way so that audio applications, consisting of sequential or parallel chains of effects, can be described in a higher level of abstraction. We have designed a task allocation tool that uses the higher level description of the audio application, consisting of effects and connections, and decides how they are mapped to the cores in the T-CREST platform. The tool then creates header files that contain information about the audio application (effect allocation, required effect connections, buffer sizes...). These header files are compiled together with the main audio processing program, which consists of a set of libraries that define the parameters and methods of each DSP algorithm, the API for the audio interface, and the NoC message passing API.

The individual effects can be seen as tasks. The static task allocation tool follows the same order as the audio signal through the effects, and decides which task is mapped to which processor core. To do this, it must have prior knowledge of the utilization values of all the tasks to decide whether any of them can be combined in the same core: this can be done if the utilization of this processor is kept under 100%. Additionally, the tool establishes the required connections between cores. After the off-line allocation, the task distribution is static during execution. If two or more tasks are combined in a single processor, the computation is done in an interleaving way, where the processing of each task is alternated with the others. This leads to a simple and clear approach where no complex scheduling is needed, therefore reducing the overheads due to context switching.

B. Processing and Intercommunication

It is common in digital audio signal processing to combine algorithms which operate on a single audio sample, such as filters or delays, with others which operate on blocks of samples, such as the fast Fourier transformation (FFT). The T-CREST audio processing platform supports combining effects with different data rates. This feature provides the system with flexibility and a high scalability in terms of the effects that can be added to the platform.

As previously mentioned, the audio effects are connected to each other forming sequential or parallel chains. In sequential chains, the data dependencies are clear, as effect $i+1$ will need the results of effect i to start processing. However, the available parallelism of the multicore platform can be exploited using a pipelining approach. This means that effects i and $i+1$ can process different data packets simultaneously, achieving computational parallelism.

The effects in the multicore platform are connected through virtual channels, which can be NoC channels, when the connected effects are on different cores, or same-core channels, when the effects are allocated in the same Patmos processor. Each NoC channel has send and receive buffers on its ends (i.e., in the SPMs of the cores it connects), which must be of the same size. On each NoC sending operation, the whole buffer of samples is transferred from the sender to the receiver. The channels in the same core are represented by memory locations: in this case, the sender and receiver tasks simply agree on a SPM address to exchange the audio data packets.

The size of the send and receive buffers affects the performance of the system directly. The buffers need to be at least as big as the data rates of the effects that the virtual channel connects. As an example, if a FFT effect processes a block size of 128 samples, then the buffers connected to this effect need to be at least 128 samples in size. However, keeping the buffer sizes to the minimum increases the amount of NoC send/receive operations done per sample, which means higher overhead due to data transfers between cores. Larger buffer sizes reduce this overhead as fewer transfers are executed, but this has higher memory requirements and increases the latency of the audio signal.

The goal is to define the buffer sizes in a way that ensures balance between the following three constraints:

- Memory requirements: the send/receive buffers are located on the SPM, which has a limited size.
- NoC transferring overhead: execution time increases when more NoC send operations are performed.
- Latency: the overall latency of the audio signal is directly proportional to the buffer sizes of the effects in the system.

For this purpose, the overhead reducing factor (ORF) term is introduced. The ORF defines by how much the buffers of each effect are increased, compared to the minimum required. Continuing with the same example as above, if a ORF of 4

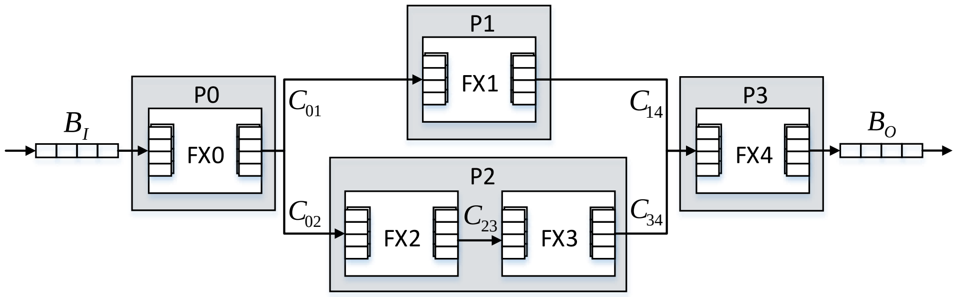

Fig. 5. Example of audio effects FX0 to FX4 mapped to cores P0 to P3 on the T-CREST platform, showing the communication channels between effects, C_{xy}. The input and output buffers of the audio interface are also shown.

is applied, the 128-sample block FFT will have send/receive buffers of 512 samples. This means that the NoC transfer overhead is reduced by 4, but the audio latency added by this larger buffers also increases by 4. The ORF of each effect needs to be set depending on the data rate of the effect: it shall be smaller for effects with higher data rates, and larger for single-sample effects.

Each effect can be seen as an individual processing unit with a receive buffer (connected to the NoC or to another effect in the same core), a processing function and a send buffer. The execution steps performed in all effects are the same:

- First, the effect receives the audio data on its receive buffer from the previous element in the system, which might be more than one if the effect is a join (i.e., it merges parallel paths).
- The samples are then processed, following the DSP algorithm that the current effect implements. When this is done, the processed samples are stored in the send buffer.
- Finally, the effect sends the data from its send buffer to the next element in the system, which again might be more than one if the effect is a fork (i.e., it splits the signal into parallel paths).

This modular approach, where effects are independent processing units that can be mapped onto different cores, together with the possibility to map multiple effects to the same core, provides very high level of flexibility and scalability of the audio processing system, since it allows to exploit parallelism and to properly combine effects to maximize the utilization of each core of the platform.

An example of a possible set of effects is shown in Figure 5, where effects FX0 to FX4 are shown with their receive and send buffers. These effects have been statically mapped to Patmos processors P0 to P3. The input buffer B_I is located in the audio interface and receives the input audio signal, which is then processed in effect FX0. After that, the signal is split into two parallel paths, which makes FX0 a fork effect. The parallel signals are processed separately, and then merged together in the join effect FX4. Finally, the audio samples are stored in the output buffer, B_O, until they leave the system. FX2 and FX3 are allocated in the same core. The communication channel C_{23} is a same-core channel, while the rest are all NoC channels.

The effects in the system use the flow-control communication paradigm to exchange data: they constantly poll requesting data on their receive buffers, and send data from their send buffers as soon as it is available. The processors do not get stuck at sending because multiple buffers are available in the same NoC channel, which means that the processor can perform a NoC send operation again, even if the previous one was not yet completed. In this case, the data is simply written to the next available buffer. Flow-control communication was chosen due to the variability in execution time of the effects, which depends on multiple factors, such as instruction and data cache hit/miss rates. This communication paradigm provides the system with elasticity, as the audio samples might be located in different elements of the platform at different times.

The audio signal flows from processor to processor through the Argo NoC. The TDM scheduling, which provides communication guarantees within a certain time bound, makes the Argo NoC a very suitable intercommunication resource for real-time audio processing, as the latency of the NoC can be precisely pre-estimated. Moreover, Argo allows overlapping of communication and computation: this is because when a Patmos processor sends a packet of samples n to the NoC, it can directly start processing the next group of samples, $n+1$, while the packet n is still traveling through the NoC.

Argo provides all-to-all communication, which means that equal bandwidth channels are available between all processor cores in the system, 4 in this case. The NoC bandwidth is the amount of packets transferred per TDM period. In the 4-core version, the TDM period is equal to 18 cycles, which is $125ns$ for the $80MHz$ source clock. Each packet consists of 64 bits of data. Therefore, the bandwidth of each Argo channel is $64/0.125 = 512Mbits/s$. As we use internally 32-bit samples, this results in a bandwidth of 16 million samples per second.

C. Latency

The latency of the audio signal can be defined as the difference in time between the moment when the first sample is output from the system and the moment when the first sample was input: it is therefore equivalent to the amount of samples that are inside the system at all times, when measured in sampling periods. The real-time audio processing T-CREST platform is designed to control and predict the latency, to ensure that it does not exceed a given limit, which is 15 ms in this case. The latency is statically set for each setup of effects: the WCET of all the effects and the worst-case NoC transfer time are known, and this allows to calculate the maximum time that it can take for a sample to travel and be processed inside the platform. The decided latency of the system must be greater or equal to this time value. In our case, the chosen latency is the minimum required.

There are 3 factors that contribute to the processing time, and therefore to the minimum latency: they are the audio interface buffers (L_{IO}), the processing of the effects (L_{FX}), and the NoC transfers (L_{NoC}). The minimum latency (L_m) is shown in equation 4.

$$L_m = L_{IO} + L_{FX} + L_{NoC} \tag{4}$$

- L_{IO} can be described as an 'intentional' latency: this is because the input buffer, of size N, is allowed to get full before processing starts (i.e., N samples are input in the system initially, before computation starts). Therefore, $L_{IO} = N$. This is done so that the samples can move among the elements of the system as explained, providing the system with elasticity.
- L_{FX} is caused due to processing applied to the signal, and depends on the buffer sizes of the effects in the setup. For any effect in the system, it is assumed that the worst-case processing time per sample is equal to the sampling period, because this is a requirement for real-time audio processing, as explained in Section III-B. This means that if an effect processes blocks of n samples, it will take as long as n sampling periods in the worst-case. This allows us to relate the latency of effect computation to samples, instead of to execution time. The total latency of each effect, measured in samples, is equal to the size of its send buffer: if the send buffer is, for example, of size $4n$, it will take a maximum of $4n$ samples until the buffer is transferred to the next element, so this is the accumulated latency. To know the total latency of all the effects, the sizes of the send buffers need to be summed. For effects that are mapped to the same core, only the last of all send buffers needs to be taken into account (the one that sends to the NoC). For parallel signals, only the longest one needs to be considered. In the example of Figure 5, the effect buffering latency would be $L_{FX} = B^o_{FX0} + max(B^o_{FX1}, B^o_{FX3}) + B^o_{FX4}$, where B^o_{FXi} is the send buffer size of effect i in samples. Only the maximum between B^o_{FX1} and B^o_{FX3} needs to be considered.
- L_{NoC} happens because the transfer of data from source to destination through the Argo NoC is not instantaneous. The latency value is derived from the worst-case packet latency of the NoC, which is equal to the TDM period: 18 cycles in the all-to-all schedule. In other words, the worst-case latency to transfer one packet from one processor to another is 18 clock cycles. A packet might contain one or two audio samples, so, in the worst case, we consider that it transfers a single sample. The worst-case latency for transferring the whole message buffer between two cores i and j through the NoC channel C_{ij} is then $18 \cdot B_{ij}$, where B_{ij} is the size of the NoC channel buffer measured in samples. The latency units can be converted from clock cycles to sampling periods by dividing the value by 1536, the sampling period. To calculate the latency of the NoC, the longest path of NoC channels needs to be considered when parallel paths are found. If we refer to the example of Figure 5 again, the total latency of the NoC is $L_{NoC} = max(L_{C_{01}} + L_{C_{14}}, L_{C_{02}} + L_{C_{34}})$. $L_{C_{23}} = 0$, as it is a same-core channel.

VI. Evaluation

This section evaluates the T-CREST audio processing platform in terms of its real-time behavior. First, the WCET of all the implemented effects are analyzed, and the variability of execution time for different buffer sizes is also shown. Finally, we present a comparison between the expected and measured latency for different combinations of effects.

A. WCET Analysis of Audio Effects

The Platin time-analysis tool, introduced in Subsection III-D, has been used to analyze the WCET bounds of all the implemented effects individually. This analysis is needed to ensure that the Patmos processor is capable of processing each one of these effects in real time. The Platin tool uses a configuration file containing information about the processor in Platin meta-information language format. In this configuration file, the most important parameters that affect to execution time are stored, which are the cache sizes and associativity values, and worst-case memory access times among others. Platin uses this information to analyze WCET of a given function, combining the trace analysis with the memory access delays.

For the evaluation, the processor and the model used in Platin are configured with data and stack caches of 4 KB each, and a method cache with 8 KB and 16 methods. For both the WCET analysis and the experimental measurements, cold and warm cache situations are analyzed, as execution time strongly depends on it. The execution time required to process a single audio sample is measured between the instant when processing of each effect starts, until it finishes (the processed samples are located in the send buffer).

Table I shows the results of the WCET analysis of the audio effects, which are compared to experimental measurements. Column 2 and 3 show WCET analysis results of the effects with cold cache and with warm cache. Column 4 and 5 show execution time measurements with cold cache and with warm cache. The last column shows the processor utilization, which is computed by dividing the WCET values (warm cache) by the sampling period of 1536 clock cycles, as shown in Equation 3.

The results presented in Table I verify that the execution time is always larger when caches are cold, due to the stalls caused by main memory access. Table I also shows that, as expected, the measured execution times are smaller than the values given by Platin (except for the cold cache values of the filter effect, which could be due to small precision differences between the hardware processor and the software model).

TABLE I

WCET ANALYSIS USING THE PLATIN WCET ANALYSIS TOOL AND EXPERIMENTAL EXECUTION TIME MEASUREMENTS OF ALL THE EFFECTS FOR A SINGLE SAMPLE, MEASURED IN CLOCK CYCLES. COLD AND WARM CACHE SITUATIONS ARE SHOWN. THE UTILIZATION OF EACH EFFECT IS ALSO PRESENTED, DERIVED FROM WARM WCET ANALYSIS.

Effect	Platin WCET (CC) cold cache	Platin WCET (CC) warm cache	Experimental meas. (CC) cold cache	Experimental meas. (CC) warm cache	Utilization (%)
Filter	4999	475	5058	336	30.9
Tremolo	2167	331	207	84	21.6
Vibrato	3514	837	441	252	54.5
Delay	3646	808	1749	504	52.6
Wah-wah	6292	846	4192	336	55.1
Chorus	5127	1029	1369	336	67.0
Overdrive	1874	113	1518	84	7.4
Distortion	3714	1014	3097	1008	66.0

Finally, the utilization values of Table I confirm that real-time processing of all the effects is possible, as the values are all below 100%. The WCET values used to calculate the utilization correspond to the Platin analysis with warm caches, as this is the stationary situation during execution (due to static allocation, each effect is continuously processed in the same processor, so its relevant data and instructions will be kept in the local caches). As the first column of Table I shows, cold cache WCET values exceed the sampling period of 1536 cycles. However, cold cache situations happen very rarely while processing: only in the very first time, or when there is a context switch. In those situations, the flow-control communication paradigm that we use ensures that there are enough samples in the output buffer, so the output audio data stream is not interrupted.

The plot shown in Figure 6 presents measurements of execution times of 4 effects with different receive/send buffer sizes, all of them in warm cache situations. The execution time values are shown per sample. The same cache parameters as mentioned before are used, and processing happens in a single Patmos core as well. However, here we measure not only the time required for the DSP algorithm computation of the effect, but also the NoC receiving and sending operations, which each processor needs to perform together with the algorithms when it processes audio in real time. This is why the values for a single sample buffer are all greater than the warm cache values of Table I. The NoC transfers cannot be analyzed with the Platin tool because it only contains a model of a single processor, but not of the multicore platform with the NoC. That is why the values shown are measured experimentally.

Figure 6 shows the reduction of execution time per sample that is caused when the ORF increases. In this case, all the effects process a single sample, so the minimum buffer size required is 1 sample. It is shown how different ORFs, up to 16, reduce the execution time per sample due to the reduction of overheads caused by NoC transfers. Table II shows this reduction for the 4 effects when an ORF of 16 is applied. The reduction is calculated as the difference relative to the maximum value, e.g., $(744-362)/744 = 0.513$ for the filter. Reductions of execution time per sample range from around 30% to around 60%, depending on the effect.

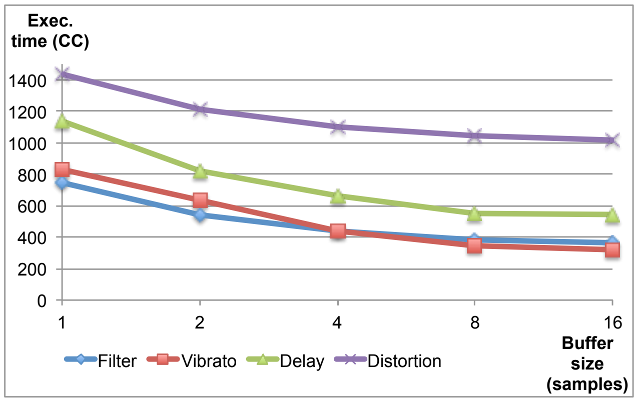

Fig. 6. Execution time per sample, measured in clock cycles, of 4 effects for different buffer sizes ranging between 1 and 16 samples.

TABLE II

EXECUTION TIME PER SAMPLE OF 4 EFFECTS WITH ORF VALUES OF 1 AND 16, AND PERCENTAGE OF REDUCTION.

Setup	Exec. time (CC) ORF = 1	Exec. time (CC) ORF = 16	Exec. time reduction (%)
Filter	744	362	51.3
Vibrato	829	320	61.4
Delay	1138	544	52.2
Distortion	1433	1016	29.1

B. Latency Verification

Table III shows the estimated and measured latency values for 4 different effect setups. Each setup is a combination of effects which can be processed in real time, as shown. The effects are connected to each other and mapped to cores in the platform, as decided by the static task allocation tool. It is not

TABLE III
ESTIMATED AND MEASURED LATENCY VALUES, SHOWN IN SAMPLES AND IN CLOCK CYCLES, FOR 4 DIFFERENT SETUPS OF EFFECTS.

Setup	Estimated (samples)	Estimated (CC)	Measured (CC)
a	80	122880	116396
b	96	147456	141764
c	128	196608	188804
d	160	245760	237944

relevant which effects are processed and in which order, as the focus here is on the latency. The estimated latency of each setup has been pre-calculated theoretically, given the buffer sizes and NoC communication requirements, following the equations presented in Subsection V-C. It shall be noted that the latency of the audio interface buffers, L_{IO}, is not considered here, but only that of the elements found within the software system. All the calculated latency values need to be rounded up to a multiple of the buffer size of the first and last effect, which is 16 samples (i.e., an ORF of 16 is applied).

Table III shows that the measured latency is not exactly the same as the estimated one, but it is in a similar range (usually, a difference up to some thousands of cycles is found). This is due to the flow-control communication used, where the elements of the system do not exchange data at constant rates, but rather as soon as it is available. What we measured in the last column of Table III is the difference between the point in time when a sample was read from the input buffer of the audio interface, and the point when the same sample was written to the output buffer. We can verify from the table that this latency is in a similar range as the estimated value, always a little bit below it. That means that the sample is written a little earlier than expected in the output buffer, and then it will reside there until it leaves the system. Even with this small divergence, it is verified that the samples arrive at the output buffer before they have to leave the platform, so the system is able to run at the pre-estimated latency.

VII. CONCLUSION

This paper presented a real-time audio processing multicore system based on the T-CREST platform, where the available parallelism of the multicore system allows processing multiple audio effects simultaneously. The platform is flexible, as effects with different data rates can be connected in the desired setup, and scalable, because new effects or processing cores can be integrated into the system.

The main concern of our design was on the time predictability of the system, thus the focus on WCET analysis, and on the latency of the signal. We have implemented a set of effects that can be connected to each other and processed in real time. The latency of the signal can be pre-calculated, and the parameters of the system can be tuned to control the latency.

Finally, the evaluation confirms that it is possible to process multiple effects with different interconnection requirements in real-time, and that the system is able to run at the pre-estimated latency.

SOURCE ACCESS

The source code of the work presented in this paper and the full T-CREST platform is available at *https://github.com/t-crest/*. The entire work is open-source under the terms of the simplified BSD license.

REFERENCES

[1] E. Battenberg, A. Freed, and D. Wessel, "Advances in the Parallelization of Music and Audio Applications," *Proceedings of the International Computer Music Conference*, pp. 349–352, 2010.

[2] Y. Wang, X. Zhu, and Q. Fu, "A low latency multichannel audio processing evaluation platform," in *Audio Engineering Society Convention 132*, Apr 2012.

[3] D. Sanz Ausin, *Audio Processing on a Multicore Platform*. Technical University of Denmark, Dept. of Applied Mathematics and Computer Science, 2017, available online at http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6966/pdf/imm6966.pdf (last accessed: Apr. 2017).

[4] D. Sanz Ausin and F. Goerge, "Design of an Audio Interface for Patmos," 2016, available online at https://arxiv.org/abs/1701.06382 (last accessed: Apr. 2017).

[5] H. L. Muller, "Designing multithreaded and multicore audio systems," in *Audio Engineering Society Conference: UK 24th Conference: The Ins & Outs of Audio*, Jun 2011.

[6] A. Partzsch and U. Reiter, "Multi core / multi thread processing in object based real time audio rendering: Approaches and solutions for an optimization problem," in *Audio Engineering Society Convention 122*, May 2007.

[7] T. Leidi, T. Heeb, M. Colla, and J.-P. Thiran, "Model-driven development of audio processing applications for multi-core processors," in *Audio Engineering Society Convention 128*, May 2010.

[8] J. A. Belloch, B. Bank, L. Savioja, A. Gonzalez, and V. Välimäki, "Multi-Channel IIR Filtering Of Audio Signals Using a GPU," *Proceedings of the IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP)*, 2014.

[9] T. Leidi, T. Heeb, M. Colla, and J.-p. Thiran, "Event-driven real-time audio processing with GPGPUs," in *Audio Engineering Society Convention 130*, May 2011, pp. 1–10.

[10] N. Jillings and Y. Wang, "Cuda accelerated audio digital signal processing for real-time algorithms," in *Audio Engineering Society Convention 137*, Oct 2014.

[11] U. Zöler, *DAFX: Digital Audio Effects*, 2nd ed. John Wiley & Sons, Inc., 2011.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the Ieee*, vol. 75, no. 9, pp. 1235–1245, 1987.

[13] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[14] T. Technologies, "DE2-115 User Manual," 2010.

[15] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch, "Patmos Reference Handbook," Tech. Rep., 2017, available online at http://patmos.compute.dtu.dk/patmos_handbook.pdf (last accessed: Apr. 2017).

[16] W. Microelectronics, "WM8731 Audio Codec," Tech. Rep., 2004, available online at http://www.cs.columbia.edu/~sedwards/classes/2012/4840/Wolfson-WM8731-audio-CODEC.pdf (last accessed: Apr. 2017).

[17] R. B. Sørensen, L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A resource-efficient network interface supporting low latency reconfiguration of virtual circuits in time-division multiplexing networks-on-chip," *Journal of Systems Architecture*, vol. 74, pp. 1–13, 2017.

[18] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit - the t-crest approach for compiler and wcet integration," in *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.