DECLARATIVELY PROGRAMMABLE ULTRA LOW-LATENCY AUDIO EFFECTS PROCESSING ON FPGA

Math Verstraelen, Jan Kuper, Gerard J.M. Smit

Computer Architecture for Embedded Systems
University of Twente
Zilverling 4078, P.O. Box 217, 7500 AE Enschede, the Netherlands
Email:{M.J.W.Verstraelen,G.J.M.Smit,J.Kuper}@utwente.nl

ABSTRACT

WaveCore is a coarse-grained reconfigurable processor architecture, based on data-flow principles. The processor architecture consists of a scalable and interconnected cluster of Processing Units (PU), where each PU embodies a small floating-point RISC processor. The processor has been designed in technology-independent VHDL and mapped on a commercially available FPGA development platform. The programming methodology is declarative, and optimized to the application domain of audio and acoustical modeling. A benchmark demonstrator algorithm (guitar-model, comprehensive effects-gear box, and distortion/cabinet model) has been developed and applied to the WaveCore development platform. The demonstrator algorithm proved that WaveCore is very well suited for efficient modeling of complex audio/acoustical algorithms with negligible latency and virtually zero jitter. An experimental Faust-to-WaveCore compiler has shown the feasibility of automated compilation of Faust code to the WaveCore processor target.

Keywords: ultra-low latency, zero-jitter, coarse-grained reconfigurable computing, declarative programming, automated many-core compilation, Faust-compatible, massively-parallel

1. INTRODUCTION

Modeling of physical/acoustical phenomena as a methodology to build electronic musical instruments has become increasingly popular since digital electronics became sufficiently powerful and cost-effective. Nowadays these models are often the mathematical core of products like synthesizers or sound effects gear (e.g. guitar-effects). Moreover, such physical models are increasingly being used during the development of acoustical music instruments [1], or used within hybrid musical instruments (like the hybrid piano). The General-Purpose Processor (GPP) in stand-alone computer systems, or in its embedded form in tablet computers or smartphones, has steadilly gained processing capacity during the past years. This has resulted in the fact that the GPP is often used as audio processing device. Consequently, audio/acoustical models are usually developed with a C-based approach. A few disadvantages of the application of the GPP are varying/unpredictable/long latency in the processing chain, the high power consumption and a limited processing capacity which limits the applicability of a GPP for complex physical modeling. Other processor technologies, like Field-Programmable Gate Arrays (FPGA), are an alternative for these types of complex modeling problems.

2. SCOPE OF THE WORK AND RELATED PROBLEMS

The scope of our work is to develop scalable (i.e. parallel computing) and low-latency processor technologies within the domain of physical modeling. State-of-the art GPP processor architectures are multi-core based. It is however a difficult task to exploit full parallelism, because it is not trivial to compile a conventional C-based program to a multi-core platform. Likewise, the usually shared and cached memory hierarchy adds another complexity level (data coherency among parallel processes, variable latencies and performance penalty when moving data between processors through a unified cached memory hierarchy). FPGAs are often used when low-latency and high-performance are crucial requirements. FPGAs offer flexibility and can be designed such that they offer the exactly required performance. However, creating an FPGA design is a specialistic task and differs from software design in many aspects. Moreover, FPGAs are not by definition flexible in the sense that the functionality can be changed easily/rapidly. As a result, an FPGA design is usually heavily optimized towards a specific task. In the field of physical modeling an example is a physical/acoustical model of a banjo instrument, based on Finite Difference Modeling [1]. Coarse-grained reconfigurable architectures (CGRA) aim to address the programmability problem of the bit-level configurable FPGA. CGRAs are usually based on regular matrices of configurable Processing Units (PUs). Usually a data-flow graph is mapped on a CGRA where the arithmetic functions map on the PUs and the communication on the interconnect network of such an architecture.

3. MAIN CONTRIBUTIONS

We have developed a CGRA, called WaveCore [2]. The WaveCore architecture is declaratively programmable through an explicit description of an algorithm in the form of a data-flow network. This data-flow network is automatically partitioned and mapped on the WaveCore architecture. The semantics of a WaveCore data-flow graph are conceptually close to properties of functional programming languages, like Faust [3]. Therefore, compilation of an algorithm which is described in a functional language towards a WaveCore data-flow graph is feasible. The WaveCore architecture is implemented as a softcore, which can be either mapped on FPGA or ASIC (Application Specific Integrated Circuit) technology. The scalability of the regular architecture, combined with the declarative and scalable programming methodology, results in the ability of automated partitioning and mapping of a data-flow graph on the architecture. A mapped algorithm behaves fully predictable, which means that stream buffering can be kept to a minimum. This

results in ultra-low processing latency. The WaveCore processor technology is optimized to characteristics which are dominant in physical modeling of audio systems (e.g. delay-lines). In this paper we will first focus on the WaveCore architecture, related to the programming methodology. Then we will explain how two example audio effects (flanger and auto-wah) are described and mapped on the WaveCore processor architecture, and how a reverberation effect which is described in Faust [3] can be automatically compiled to WaveCore. Next we will present a benchmark modeling algorithm, and highlight the efficiency. Finally we draw some conclusions and we give some directions to future work.

4. WAVECORE PROCESSOR TECHNOLOGY

Like we mentioned, WaveCore is a CGRA. This architecture consists of a cluster of interconnected Processing Units (PU). These PUs are small RISC (Reduced Instruction Set Computer) processors with a dedicated instruction-set which is specifically optimized towards audio/acoustical modeling. The WaveCore programming methodology is based on a native WaveCore programming language, and based on a declarative description of a hierarchical network of processing primitives. Despite the fact that the WaveCore language is native, the structure of the language matches closely with existing functional languages, like Faust [3]. The WaveCore processor architecture is implemented as a technology independent and configurable soft-core (VHDL), which as a prototype has been targeted to a commercially available FPGA development platform (Digilent Atlys, [4]). A WaveCore cluster and associated mapping tool can be automatically generated. In the following sections we will first outline the WaveCore programming methodology, followed by the associated WaveCore PU cluster architecture.

4.1. Programming model

The WaveCore programming model is based on explicit description of a data-flow-graph in a declarative manner. An example of such a graph is depicted in fig. 1. At top-level the graph consists of one or more 'actors' (i.e. processes) which are interconnected by means of 'edges'. An actor can have multiple inbound (i.e. input) and outbound (i.e. output) edges. Data is carried across the edges, where a data-packet is called a token. Each edge is associated with a predefined token-type. For example: an edge might represent a stereo audio channel, carrying tokens which consist of two floating-point numbers at a token rate of 48kHz. Each edge in the graph has a programmable token buffering capability. The example graph in fig. 1 represents an audio processing application. This application is controlled by an actor called "run-time control actor". This actor might run on a host processor and communicates with a WaveCore actor (which we call a WaveCore Process (WP)) through control edge E2. This edge E2 carries control tokens which represent for instance audio effects settings, like phasing depth. The "audio interface actor" within the example application graph represents an intermediate process between an audio codec (e.g. AC97 device) and the WP. This audio interface actor communicates with the WP through two edges: E1 which carries the input tokens(s) to the WP and E2 which carries the processed WP output tokens. The actual signal processing algorithm runs on two example WPs, called WP1 and WP2. WP1 on its turn consists of two WP-partitions WP1.a and WP1.b. Ultimately the WP, or WP-partition is composed of "primitive actors" (PA). As such, the

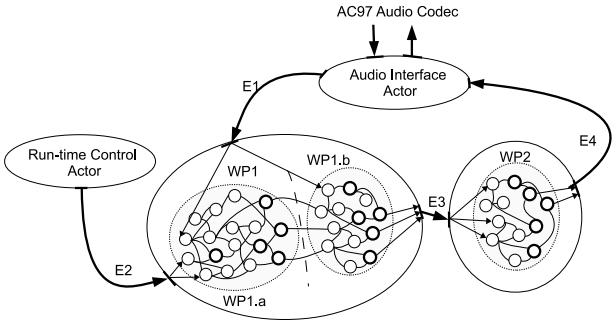

Figure 1: *Data-flow graph*

PA is the algorithmic primitive within the declarative WaveCore programming methodology. The PA is defined in the next subsection. Each actor in the graph is executed (i.e. fired) periodically, where each actor might be fired with a different (through coherently related) frequency. Within WaveCore, we chose to apply a centralized scheduler which orchestrates the firing of all the actors in the graph. The reason that we chose for this scheduling principle, rather than a purely data-flow driven schedule which is more common in data-flow architectures, is that a centralized scheduling principle yields a fully predictable and jitter-free execution of the overall graph. As a result, large closed-loop graphs with a relative large number of actors still yield a fully predictable overall execution when applying this centralized scheduler principle.

4.1.1. Primitive Actor

Like explained, the Primitive Actor (PA) is the basic processing element in the WaveCore programming methodology. The PA is depicted in fig. 2. The definition of the PA is based on fundamental discrete-time audio processing characteristics. Besides the common basic mathematical operations like addition, subtraction, multiplication, etc. a dominant property is the delay function. Delay-lines are dominantly present in many audio and acoustical modeling algorithms [5], like reverberation, string modeling etc. Furthermore, dynamic delay-line length variation is an additional basic property which is also present in many modeling phenomena (like Doppler shifting, or multi-path interference with time-modulated path-length such as "flanging"). The WaveCore PA is depicted in fig. 2. The PA has at most two inbound edges $x_1[n]$ and

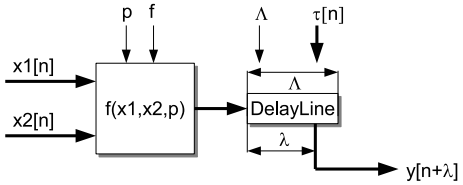

Figure 2: *WaveCore Primitive Actor (PA)*

$x_2[n]$ and one outbound edge $y[n + \lambda]$. There are different types of PAs which are indicated with the function identifier f (see table 1) A number of PAs (like the MUL-type PA) need a parameter p. Each PA is associated with an optional delay-line which is automatically inferred when the delay-line length Λ is greater than zero. The effective length λ of the delay-line can be run-time var-

ied with the inbound edge $\tau[n]$ according to the following relation:

$$1 < \lambda[n] < \Lambda \tag{1}$$

According to:

$$\lambda[n+1] = \lfloor \Lambda.(\tau[n] - 1) \rfloor \tag{2}$$

With:

$$1 < \tau[n] < 2 \tag{3}$$

The programmer should take care that $\tau[n]$ is bounded within the specified range. Hence, the behavior is not specified outside the specified range.

Table 1: *PA types*

PA type	Mnem	Function
Adder	ADD	$y[n+\lambda] = x_1[n] + x_2[n]$
Comparator	CMP	$y[n+\lambda] == (x_1[n] > x_2[n])$
Divider	DIV	$y[n+\lambda] = x_1[n]/x_2[n]$
Logic funct.	LGF	$y[n+\lambda] = LogFunc(x_1[n], x_2[n], p)$
Look-up	LUT	$y[n+\lambda] = Lookup[\lfloor Scale(x_1[n]) \rfloor]$
Multiplier	MUL	$y[n+\lambda] = x_1[n].x_2[n]$
Mul/Add	MAD	$y[n+\lambda] = p.x_1[n].x_2[n]$
Amplifier	AMP	$y[n+\lambda] = p.x_1[n]$
Noise Gen	RND	$y[n]$=$p.rnd < 0 : 1 >$

4.2. Processor architecture

The block diagram of an embedded WaveCore PU cluster instance is depicted in fig. 3. The cluster in fig. 3 consists of 5 interconnected PUs, and a shared local memory. Each PU represents a small IEEE754 compliant single-precision floating-point based RISC processor. The PU is optimized to sequentially fire all PAs in a WP-partition. If a WP is partitioned into two or more WP-partitions, then the data which is associated to the graph-cut(s) is streamed over the PU interconnect network which is called the GPN (Graph Partition Network). The memory hierarchy spans three levels and does not contain any caching. Level1 implies PU-proprietary tightly coupled memory within each PU instance. Level1 memory is used for in-place execution of WPs or WP-partitions (scratch memory and storage of state-variables) Level2 embodies a shared PU cluster memory. Level3 memory is located outside the PU cluster and is usually an off-chip bulk memory like DDR. WP token data buffer space is either allocated in Level2 (for spatially close WPs), or Level3 memory. Delay-line buffer space is also mapped onto Level2 or Level3 memory. The memory hierarchy has been designed in such a way that the locality of reference principle is maximally utilized. This implies that the data-traffic between the levels is kept to a minimum. The instruction set of the PU is optimized in such a way that it is enabled to fire a PA within a single instruction, including all the necessary memory references, pointer updates, arithmetic operations etc. Moreover, special measures have been taken to hide Level2/Level3 memory latency for the PU.

Like we mentioned in the introduction, all the actors in the data-flow graph are fired by means of a centralized scheduler. This means that all the PUs in the cluster are triggered by this centralized scheduler. The overall PU cluster architecture has been designed in such a way that the real-time constraints are always met. The cluster can be initialized through the "Host Processor Interface" (HPI), which provides access to instruction memory, data memory and registers for all the embedded PUs. Similarly, the HPI is used for run-time control of the WaveCore application. Run-time application control can either be performed by direct PU level1 memory write-access, or via control tokens through Level3 memory.

4.3. WaveCore FPGA development platform

We have generated an optimized WaveCore PU cluster for a commercially available Digilent Atlys FPGA platform. The heart of this platform is a Xilinx Spartan6 LX45 FPGA. The platform contains a rich variety of interfaces (USB, audio, HDMI, etc.) and memory. The WaveCore/Atlys architecture is depicted in fig. 3. This architecture uses the on-board AC97 codec, obviously the FPGA, the USB interface and DDR2 memory. The generated WaveCore PU cluster consists of 5 PUs and 16kByte Level2 memory.

The feasible clock frequency for the PUs is largely dependent on the target technology, in this case the Spartan6 LX45 FPGA. For this typical device we obtained a PU clock frequency of 86MHz (which equals 1792 times the programmed AC97 audio rate of 48kHz). Given the fact that a PU can execute a PA within a single instruction and it is a fully pipelined RISC processor architecture, this yields a processing capacity of 1792 PAs per audio sampling period per PU at 48kHz audio rate. Hence, the PU cluster on the Atlys board has a total processing capacity of 8960 PAs. This is equivalent to 860 MFLOPs sustained performance.

The PU cluster is embedded in a FPGA SoC (System on Chip) topology, as can be seen in fig. 3. The cluster has its own dedicated port to the embedded DDR2 controller, which provides access to the 128MByte DDR2 memory (level3 memory). The AC97 codec chip on the Atlys board streams directly into the DDR2 memory through the "Stream Actor" within the "Digital Audio Interface" on the FPGA. The system is controlled through the USB interface. Hence the host processor is supposed to be an external device like a notebook, smartphone (through BT), etc. The host processor is enabled to initialize/reconfigure the PU cluster and to control the application at run-time (e.g. virtual knobs).

Mapping of the example data-flow graph in fig. 1 onto the WaveCore PU-cluster in fig. 3 is straightforward. The "Run-time Control Actor" runs on the host computer (e.g. notebook). This control process can either pass control tokens into a dedicated buffer in DDR2 memory (represented by E2, see fig. 1), or directly write into PU level1 memories. The "Audio Interface Actor" runs on the FPGA as a hardwired process. This actor is basically a DMA controller which moves audio samples from the AC97 ADC (Analog to Digital Converter) to a dedicated token buffer in DDR2 memory, and audio samples from token buffer in DDR2 memory to the AC97 DAC (Digital to Analog Converter). The AC97 codec can be initialized by the host processor through USB. The actual processing is performed by the example WP1 and WP2 actors which are mapped on the WaveCore PU cluster. The number of occupied PUs within the cluster depends on the complexity of the WPs (number of PAs). It might be the case that WP1.a, WP1.b and WP2 run on three PUs, but these three WP(partitions) might also be mapped on a single PU if these are sufficiently small. The token

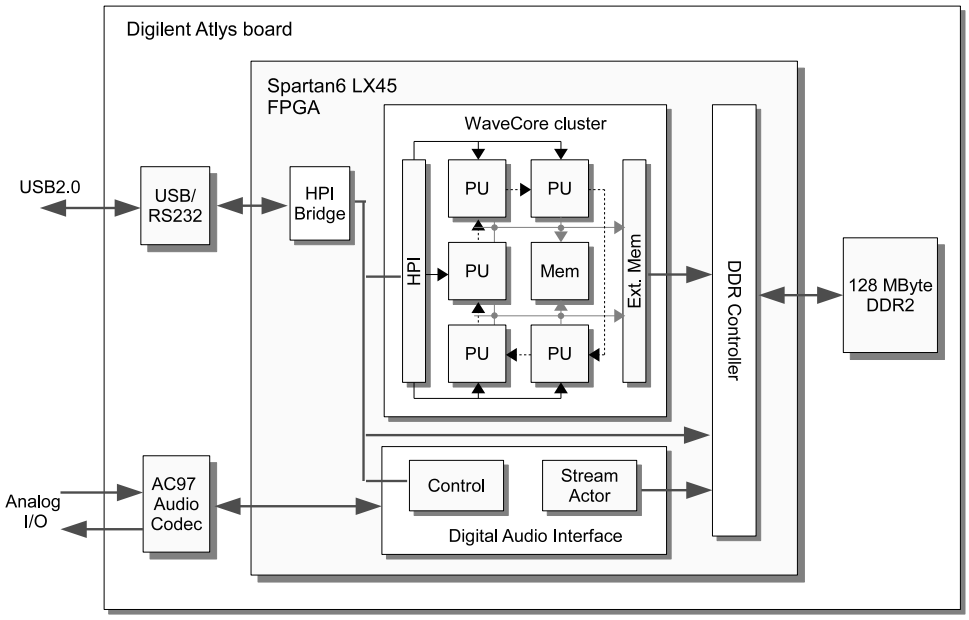

Figure 3: *WaveCore development platform architecture*

buffering for E3 is mapped on the locally embedded level2 memory within the cluster. Scheduling/firing of the actors is driven by the AC97 codec, which dictates the token frequency. The scheduler is embedded in the "Audio Interface Actor" and periodically fires the "Streaming Actor" and mapped WPs on the PU cluster through the HPI interface. Note that the "Run-time Control Actor" typically fires only incidentally, or at least at a rate which is far lower than the audio token rate of 48kHz.

4.4. FPGA mapping

The WaveCore SoC design (PU-cluster and infrastructure) in fig. 3 is mapped on the Xilinx LX45 FPGA on the Atlys board. The WaveCore PU cluster contains 5 PUs, where each PU is dimensioned to execute WP-partitions with up to 2048 PAs. The PU clock frequency is derived from the AC97 audio clock by an on-chip PLL (Phase Locked Loop). Each PU uses 120kByte level1 memory. The level2 memory is configured to 16kByte. This WaveCore configuration requires 21% of the slice registers, 68% of the slice LUTs, 75% of the block-RAMs, and 51% of the DSP48 units on the FPGA. The mapping floorplan is displayed in fig. 4. Each color in the floorplan represents a WaveCore PU.

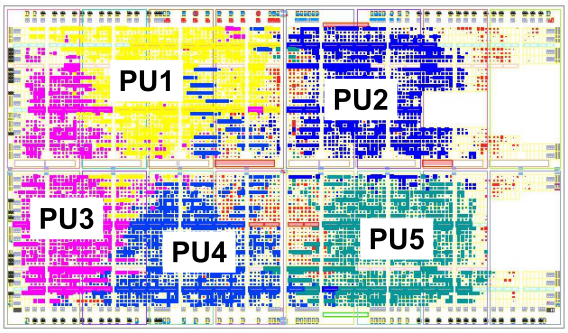

Figure 4: *Xilinx LX45 FPGA floorplan for WaveCore PU cluster*

4.5. Latency, jitter and buffering

Latency, jitter and sample buffering are closely related. Jitter is usually caused by unpredictable execution behavior. This unpredictability can be caused by several factors like cache-misses, interrupts, unpredictable round-trip delay times for shared memory read transactions or other process stalls due to shared resource conflicts. For an audio application it is unacceptable to stall the production of output samples, or to skip output samples in case of a stalled process. To prevent this, buffering is usually applied. The required buffering depth is directly related to the worst-case stall time of the processing device. Buffering however has the disadvantage that it introduces processing latency. This latency is critical in systems where part of the application runs on the processing system, and the other part communicates with the system in a closed-loop (e.g. the hybrid piano, or real-time guitar effects processing).

The processing latency within the WaveCore architecture can be as short as a few audio sampling periods (below 100 μsec @ 48kHz sampling rate). The actual processing latency depends on the token buffer sizes (number of tokens within an edge channel) and the nature of the processing algorithm which runs on the PU cluster. This latency can be that short because of the real-time guaranteed, fully predictable and jitter free execution of the overall WaveCore processing chain. Jitter-free execution is a direct result of the strict process-firing mechanism on the one hand, and guaranteed execution of WP (or WP-partitions) on the other hand. Moreover, the carefully designed memory hierarchy results in a modest load on the externally shared level2 and level3 memory resources. Finally, the static and fully predictable processing schedule prevents the need for extensive buffering.

5. AUDIO EFFECTS IN WAVECORE TECHNOLOGY

Like we explained, a WP (or WP-partition) consists of a network of interconnected PA instances. Furthermore, the available PA types enable both arithmetic as well as logical/control functionality. The abstraction level of a WaveCore WP matches closely with the level at which DSP functions at block diagram level are often specified. We will demonstrate this with two example audio effects algorithms, described in WaveCore: the 'Flanger' and 'Auto-Wah' sound effect algorithms. Additionally, we will show the results of an experimental compilation from the functional audio description language Faust to a WaveCore WP through an example algorithm: the 'Zita-Rev1' reverberator.

5.1. Flanger

The flanger audio effect is widely used within several musical instruments. The effect is based on the principle of varying multi-path acoustical wave interference. In this subsection we will focus on a WaveCore model of a flanger [5]. The core of the flanger algorithm is a delay-line which length is modulated by a Low-Frequency Oscillator (LFO). The acoustical input signal is split: one path goes through the modulated-length delay-line (path1) and the other travels without delay (path2). Subsequently, the path1 and path2 signals are added, yielding the output signal $y[n]$. The mentioned interference when varying delay-length(s) are applied yields the typical flanger effect.

The block diagram of the flanger is depicted in fig. 5. The

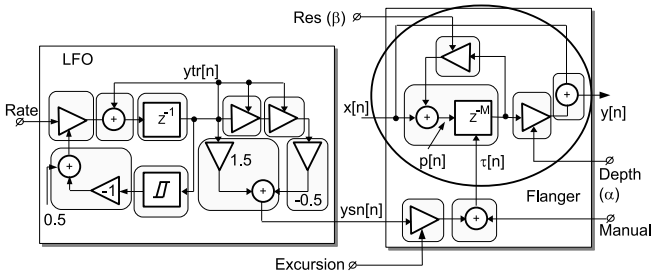

Figure 5: *Flanger WaveCore process*

flanger implementation consists of two WaveCore WP-partitions. The right WP-partition of the block diagram represents the core of the flanger. Those parts of the block diagram which are bounded by the round-edge boxes represent PAs (e.g. the adder and associated delay-line with input $x[n]$ is one PA). The primary input signal $x[n]$ is added to a delayed and scaled copy of the input signal $\alpha.x[n-\lambda]$, while a fraction β of the delay-line output is fed-back to the input of the delay-line. The actual multi-path interference takes place at the output adder where $x[n]$ is added to the delay-line output. The feedback-path, where a fraction of the delay-line output is fed back into the input of the delay-line, intensifies the effect.

In our next analysis, where we will show that the flanger core is a variant of a comb-filter, we assume λ to be constant. The difference equations for the upper-part of the flanger-core (within the ellipse) are given in equations 4 and 5 .

$$p[n] = x[n] + \beta.p[n-\lambda] \tag{4}$$

And

$$y[n] = x[n] + \alpha.p[n-\lambda] \tag{5}$$

With:

$$\lambda[n+1] = \lfloor M.(\tau[n]-1) \rfloor \tag{6}$$

After applying the z-transformation to the difference equations and merging the transformed equations, we find the transfer function in the z domain which is defined in equation 7

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1+(\alpha-\beta)z^{-\lambda}}{1-\beta.z^{-\lambda}} \tag{7}$$

In order to analyze the magnitude response of the flanger-core, we replace z by $e^{j\theta}$, and subsequently derive the magnitude response which is defined in equation 8.

$$|H(e^{j\theta})| = \sqrt{\frac{1+(\alpha-\beta)^2+2(\alpha-\beta)cos(\lambda\theta)}{1+\beta^2-2\beta cos(\lambda\theta)}} \tag{8}$$

The magnitude response of the comb filter reveals a number of equidistant peaks and notches. The location of the peaks in the magnitude response of the comb-filter follows from equation 8, and is as defined in equation 9.

$$\theta_k^{(p)} = k\frac{2\pi}{\lambda}, k = 0, 1, 2, \cdots, \lambda-1 \tag{9}$$

So, the number of equidistant notches and peaks in the magnitude response of the comb-filter is equal to λ and is proportional to $\tau[n]$. The effect of the feedback path with the res amplifier is that the peaks in the magnitude response get smaller and intenser for high res values, which also follows from equation 8.

The delay-line length λ is modulated by the left-part of the block diagram in fig. 5: the LFO WP-partition. This LFO generates an approximated sine-wave with a sub-Hz frequency, which is determined by the $rate$ parameter. The core of the LFO is an integrator with a hysteresis-PA in its feedback path, as is depicted in fig. 5. The hysteresis-PA switches symmetrically between -1 and 1 (the output of the hysteresis-PA toggles between 0 and 1), which equals the amplitude of the generated triangular wave $ytr[n]$. The multipliers and adder with input $ytr[n]$ implement a third-order polynomial, as in equation 10

$$ysn[n] = 1.5ytr[n] - 0.5ytr[n]^3 \tag{10}$$

This polynomial shapes the triangular waveform into an approximated sine-wave $ysn[n]$. The amplitude of $ysn[n]$ is attenuated with the parameter $excursion$, and an offset tau is added. Finally, the resulting signal (which oscillates within range $1 < \tau < 2$) is used to modulate the delay-line length, according to equation 2. Note that we did not take fractional delay-line length interpolation into account (to suppress "zipper-noise" due to the discrete-length model of the delay-line).

The overall behavior of the flanger is represented by the spectrogram in fig. 6. This spectrogram is a representation of the WaveCore simulation of the flanger model, with an impulse-train as input signal.

The WaveCore implementation of the flanger is a netlist representation of the block diagram in fig. 5. The applied PAs in this figure are indicated by the round-edge rectangular boxes. The parameter

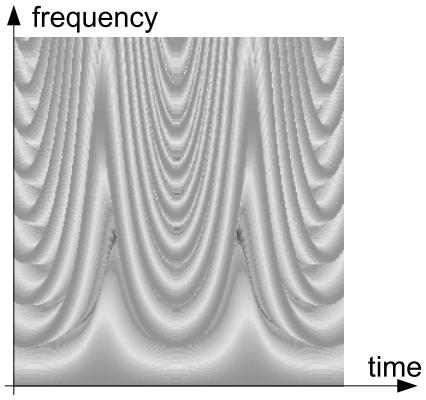

Figure 6: *Magnitude response spectrogram of flanger algorithm*

inputs ($rate, res, excursion, manual$ and $depth$) are run-time controllable, similar to the control knobs on a physical flanger device. As can be seen in fig. 5 the flanger implementation requires 16 WaveCore PAs, which implies less than 1% of the processing capacity of a single PU. The required minimum clock frequency for this application is 2.4 MHz. The PU memory requirement is 176 bytes. The processing latency of the flanger is zero, since the shortest path from $x[n]$ to $y[n]$ does not contain delay-elements. This implies that the overall flanger latency equals the latency in the AC97 codec, plus one additional sampling period for token buffering.

5.2. Auto-Wah

Similar to the flanger, the 'WahWah' audio effect is also widely used in several electronic musical instruments. The name reveals its sound effect. This sound effect is based on a bandpass filter with center frequency ω_0 and quality-factor Q, where the center frequency marks the middle of the pass-band and quality-factor determines the bandwidth of the pass-band. The center-frequency is dynamically varied within the WahWah effect. This can be done in various ways (e.g. expression pedal or controlled by input signal properties like envelope). In our example we chose an envelope controlled WahWah, which is sometimes referred to as 'Auto-Wah'. The continuous-time transfer function for a second order bandpass filter with ω_0 and Q parameters is defined in equation 11.

$$H(s) = \frac{(\frac{1}{Q.\omega_0}).s^2}{(\frac{1}{\omega_0^2}).s^2 + (\frac{1}{Q.\omega_0}).s + 1} \tag{11}$$

We need to translate the continuous-time transfer function to its discrete-time counterpart. For this purpose we use the bilinear transformation, which substitutes s with z, according to the substitution rule in equation 12.

$$s \leftarrow \frac{2}{T_s}.\frac{z-1}{z+1} \tag{12}$$

With T_s the sampling period. Application of the bilinear transformation yields the discrete-time transfer function $H(z)$ in equation 13, which embodies the required bandpass filter behavior with ω_0 and Q parameters.

$$H(z) = \frac{a(1-z^{-2})}{(a+b+1) + 2(1-b)z^{-1} + (1-a+b)z^{-2}} \tag{13}$$

With:

$$a = \frac{2}{Q.\omega_0.T_s} \text{ and } b = \frac{4}{T_s^2.\omega_0^2} \tag{14}$$

The final step is to correlate the coefficients in the derived transfer function $H(z)$ with the coefficients in the general second order discrete-time transfer function, which is defined in equation 15.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{15}$$

The relation between the system parameters in transfer function in equation 13 and the parameters in the generic second order transfer function in equation 15 is defined in equations 16,17 and 18.

$$b_0 = \frac{a}{d}\,,\, b_1 = 0\,,\, b_2 = \frac{-a}{d} \tag{16}$$

And:

$$a_1 = \frac{2(1-b)}{d}\,,\, a_2 = \frac{1-a+b}{d}, \tag{17}$$

With:

$$d = a + b + 1 \tag{18}$$

We map the generic transfer function in equation 15 on a direct-form II structure. This structure is depicted in the "DF2-IIR" WP-partition in fig. 7. The transfer function in equation 15, and associated parameters in equations 16, 17 and 18 apply to this DF2-IIR structure (note that the parameter b1 equals zero, and therefore is left out). The recipe for computing the coefficients b_0, b_2, a_1

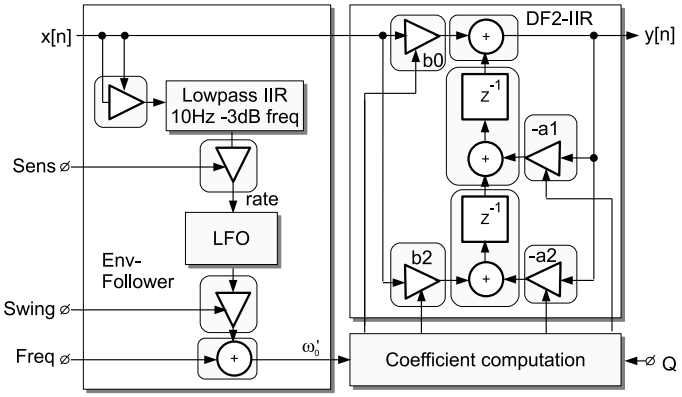

Figure 7: *AutoWah WaveCore process*

and a_2 from equations 16,17 and 18 is implemented in the WP-partition called "Coefficient Computation" (obviously with inputs ω_0 and Q). The "Env-Follower" WP-partition in fig. 7 detects the envelope of the input signal $x[n]$ and uses this envelope to compute the center frequency ω_0. Envelope detection is implemented by squaring the input signal, followed by lowpass filtering. The envelope signal is scaled and subsequently determines the frequency of the LFO (same LFO as used in the flanger in fig. 5). The sine wave shaped output of the LFO is scaled with the $swing$ parameter and added to the offset $freq$, and finally fed to the "Coefficient Computation" WP-partition.

The AutoWah WP requires 46 WaveCore PAs. This implies 2.2% of the processing capacity of a single PU. The AutoWah application requires a minimum clock frequency of 2.5 MHz. The PU memory requirement for the AutoWah is 368 bytes. The processing latency of the AutoWah algorithm is zero, since the shortest path from $x[n]$ to $y[n]$ does not contain delay-elements.

5.3. Automated compilation from Faust to WaveCore

In the third mapping example we focus on the WaveCore programming language compatibility with Faust [3]. Faust stands for Functional AUdio STream, and is a functional language designed for audio processing. The Faust compiler produces C++ code, specifically targeted to a C++ signal processor class. The Faust compiler can also produce a graph representation of the data-flow network which is extracted from the Faust language description. Typically an algorithm which is described in Faust consists of a data-flow process which is intended to run at real-time audio rate, and a control part which intends to implement the run-time control (e.g. sliders, knobs, etc.). We have developed an experimental compiler which maps the Faust graph representation onto a WaveCore process. This compiler writes a WP netlist description which subsequently can be compiled to WaveCore object code by the WaveCore mapping tool. We used the Faust implementation of a stereo variant of the Zita-Rev1 [6] reverberator to show the feasibility of WaveCore code generation from Faust. The Zita-Rev1 algorithm has been converted to WaveCore in a fully automated way.

The compiled Zita-Rev1 WP requires 284 WaveCore PAs, where 30 of these PAs infer delay-lines. This implies 12% of the processing capacity of a single PU. The Zita-Rev1 application requires a minimum clock frequency of 22 MHz. The PU memory requirement for Zita-Rev1 is 2608 bytes. The processing latency of the compiled Zita-rev1 Faust algorithm is zero since the shortest path from input $x[n]$ to output $y[n]$ does not contain delay-elements.

6. EVALUATION

We have developed a benchmark WaveCore algorithm set, which implements a combination of guitar effects processing and digital wave synthesis. This benchmark represents a guitar model (controlled by a console), a guitar-effects gear box and a model of distortion and speaker cabinet. The block diagram of this benchmark is depicted in fig. 8. The complete model is encoded in one composed WaveCore process which breaks down into four hierarchically built WP-partitions.

The first WP-partition in the chain instantiates 6 guitar string models. For a guitar string model we use a DWG (Digital Wave Guide) model, based on the Karplus-Strong algorithm [7]. Each string in the 6-string model can be tuned at run-time by a process which runs at the host computer. Moreover, plucking each individual string as well as adding damping characteristics (controlling the timbre of each individual string) can be controlled at run-time. Hence, a high level of "player" control is enabled by the model: like string bending, playing chords, artificial finger-picking etc.

The second WP-partition in the chain implements an acoustical model of a guitar body. The implementation of this model is based on a 1700-taps FIR filter. The six-string model, added with the acoustical guitar body model forms a digital model of a guitar: the "digitar".

The third WP-partition implements the effects gear-box. This is a model of a multi-effects rack. The digitized AC97 input signal is added to the "digitar" at the input of the gear model, and subsequently fed to the effects models in the rack. This enables to plug-in a real guitar into the analog input of the FPGA board. The effects in the rack are a 12-stage envelope Phaser, the "Zita-Rev1" reverberator [1] [6], the Auto-Wah, the Tremolo effect, and the flanger. The outputs of the individual effects are scaled and mixed with the unprocessed (i.e. "dry") signal which is fed through the "Bypass" unit. The effects in the gear-box, as well as the scaling/mixing is controlled at run-time by a process which runs on the host computer.

The fourth WP-partition implements a distortion model which is based an the BOSS DS1 distortion pedal [8], a speaker cabinet model and stereo rendering. The signal which is fed through the DS1 model is scaled and added with a scaled "dry" signal (through the "Bypass" unit). The output of the added signal is fed through a 1700-taps FIR filter which represents an acoustical model of a speaker cabinet. Finally, the output of the cabinet model is routed to the "left" channel output of the AC97 codec, and a delayed copy of this signal ("right") is routed to the "right" channel of the AC97 codec. The parameters for the DS1, the dry/wet mixing and the length of the stereo rendering delay can be varied at run-time by the host-computer. Table 2 shows the mapping results of the benchmark algorithm on the WaveCore/Atlys platform. The algorithm requires 365 MFLOPS and the overall processor utilization is 47% (2 idle PUs and the other ones not entirely loaded). Furthermore, the processing latency in the entire processing chain equals only two sampling periods (shortest path in the chain). The external memory bandwidth for this algorithm equals 9.6 MBytes/s, which is very modest. Hence, the real-time execution requirements of the overall algorithm are easily fulfilled with zero processing jitter.

Table 2: *Mapping results of Digitar benchmark algorithm set*

WP-partition	#PAs
6-String	184
GuitarBody	1712
GearBoxModel	551
Distortion/Cabinet-model	1774

PU	#Mapped PAs	Utilization	#Mapped DelayLines
1	738	41%	48
2	1714	95%	0
3	1775	99%	1
4	0	0%	0
5	0	0%	0

Next to the described benchmark, we also applied another experiment to fully load the WaveCore cluster with interconnected biquad chains. We found that it is possible to compile 1710 biquad filter instances, divided over 30 WP-partitions within a single WP.

7. CONCLUSIONS AND FUTURE WORK

We have developed a scalable coarse-grained reconfigurable data-flow architecture, called WaveCore. WaveCore is optimized to the

[1] The Zita-Rev1 reverberator model has been generated from a Faust source by an experimental *Faust2WaveCore* compiler

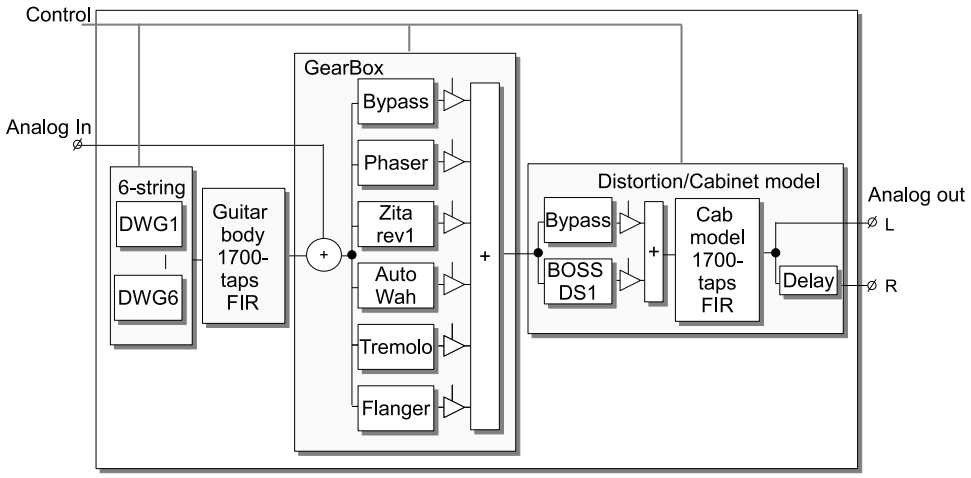

Figure 8: *Digitar benchmark setup.*

application domain of audio and acoustical processing. The programming model is declarative and based on explicit description of hierarchical data-flow networks. We found that the programming methodology is to a large extend compatible with Faust, which has been demonstrated with an experimentally compiled 'Zita-Rev1' reverberation algorithm. We have configured a WaveCore PU cluster for the Digilent Atlys, and applied this flashed WaveCore/Atlys board as a demonstrator platform.

We have developed a comprehensive digital effects processor for this platform which serves as a combination of guitar-effects processor and synthesizer with a DWG-based guitar model. The processing latency is negligible (few audio sampling periods) with zero processing jitter. The processing capacity of the demonstrator WaveCore/Atlys board proves that the architecture is very efficient.

Interesting target applications for the WaveCore technology range from cost effective audio effects solutions (e.g. multi-effects gear), to complex audio/acoustical modeling. The ultra low latency and real-time guaranteed execution makes it possible to apply the technology to hybrid instruments. The declarative programming methodology enables efficient interfacing of the technology to a class of functional languages with data-flow characteristics. Faust is an obvious example of this, but other existing data-flow programming methodologies are also interesting candidates. Functional language interfacing is an interesting topic for future work. Further benchmarking of the processor technology is also an interesting topic for future work.

8. ACKNOWLEDGMENTS

We thank Edgar Berdahl (CCRMA) for many constructive discussions on hybrid acoustical modeling, related to processor architectures. We thank Andreas Degert (Guitarix) for his contribution on the experimental *"Faust2WaveCore"* compiler which has resulted in a fully automatically compiled "Zita-Rev1" reverberation model. We also thank Jan Jacobs and Ramon Jongen from "Zuyd University of Applied Sciences" for their constructive discussions on architectures and signal processing applications.

9. REFERENCES

[1] Rolf Bader Florian Pfeifle, "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 (DAFx-12)*, York, UK, Sept. 17-21 2012.

[2] Math Verstraelen, Jan Kuper, and Gerard J.M. Smit, "Wavecore: a reconfigurable mimd architecture," Submitted for publication, 2014.

[3] Stephane Letz Yann Orlarey, Dominique Fober, "FAUST (programming language)," Available at http://faust.grame.fr/, accessed Dec. 08, 2013.

[4] Digilent Inc., "Atlys Spartan-6 FPGA Development Board)," Available at http://www.digilentinc.com/Products/, accessed Jan. 21, 2014.

[5] Julius O. Smith, Ed., *Physical Audio Signal Processing for virtual musical instruments and digital audio effects*, W3K Publishing, USA, 2010.

[6] Julius O. Smith, "Zita-rev1," Available at https://ccrma.stanford.edu/ jos/pasp/Zita_Rev1.html, accessed Jan. 21, 2014.

[7] Kevin Karplus and Alex Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, May 1983.

[8] Jonathan S. Abel David T. Yeh and Julius O. Smith, "Simplified, physically-informed models of distortion and overdrive guitar effects pedals," in *Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07)*, Bordeaux, France, Sept. 10-15 2007.

[9] M. R. Schroeder and B. F. Logan, "Colorless artificial reverberation," *Journal of the Audio Engineering Society*, vol. 9, pp. 192–197, 1961.

[10] Udo Zolzer, *DAFX: Digital Audio Effects*, chapter Chapter 2: Filters, pp. 55–56, John Wiley & Sons, Ltd, 2002.