FEASIBILITY ANALYSIS OF REAL-TIME PHYSICAL MODELING USING WAVECORE PROCESSOR TECHNOLOGY ON FPGA

Math Verstraelen[1], *Florian Pfeifle*[2], *Rolf Bader*[2]

[1]Computer Architecture for Embedded Systems
University of Twente, the Netherlands
[2]Institut für Systematische Musikwissenschaft
Universität Hamburg

ABSTRACT

WaveCore is a scalable many-core processor technology. This technology is specifically developed and optimized for real-time acoustical modeling applications. The programmable WaveCore soft-core processor is silicon-technology independent and hence can be targeted to ASIC or FPGA technologies. The WaveCore programming methodology is based on dataflow principles and the abstraction level of the programming language is close to the mathematical structure of for instance finite difference time-domain schemes. The instruction set of the processor inherently supports delay-lines and data-flow graph constructs. Hence, the processor technology is well suitable to capture both digital waveguide as well as finite-difference oriented algorithm descriptions. We have analysed the feasibility of mapping 1D and 2D finite-difference models onto this processor technology, where we took Matlab reference code as a starting point. We analyzed the scalability and mapping characteristics of such models on the WaveCore architecture. Furthermore we investigated the composability of such models, which is an important property to enable the creation and mapping of complete musical instrument models. One part of the composability analysis has been to combine digital waveguide (FDN reverberation model) with finite-difference time-domain models (primitive six-string instrument model). The mapping experiments show a high efficiency in terms of FPGA utilization, combined with a programming methodology that matches in a transparent way with the mathematical abstraction level of the application domain. We used a standard FPGA board to get full-circle confidence of the carried-out analysis. The WaveCore compiler and simulator results have shown the scalability of the processor technology to support large models.

1. INTRODUCTION

The development of sound synthesis digitization is to a large extend relying on the availability of computational power and hence indirectly linked to the progress of Moore's law. Sound synthesis itself is built upon a production model. The nature of such a model can be abstract and hence based on the production of the desired sound itself (e.g. Digital Waveguide or wavetable techniques) or based on a model that physically represents the object that produces sounds such as Finite Difference Time Domain (FDTD). A historical overview and elaborate explanation of these techniques can be found in [1], [2], and [3]. Mapping of FDTD based physical models on processors is challenging because of the huge computational requirement. Despite the impressive evolution of general purpose processors which are multi-core nowadays and widely applied in mainstream computer platforms, these processors can still not meet the computational requirements for solving FDTD models within reasonable timespans, let alone real-time. As a result, the focus is on different processor technologies, like GPGPU (General-Purpose computing on Graphics Processing Units) [4] and FPGA (Field Programmable Gate Array). Solutions based on GPGPU can achieve impressive accelleration and offer flexibility because of the programming versatility. Feasibility of real-time simulation of physical models on GPGPU has been investigated by Hsu et. al. [5]. They have been able to map square shaped 2D membrane structures with grid sizes up to 81x81 points onto a GPGPU at 41.1 kHz real-time, and compared this against GPP performance. FPGA's enable parallel computing to the extreme and have the potention to meet the real-time computational requirements. The difficulty with FPGA's, however, is programmability. In principle an FPGA is a fabric of primitive logic gates and memory resources that can be configured (interconnected) in an almost arbitrary way. This implies that an FPGA "programmer" has to develop a processor soft-core that is subsequently mapped on the FPGA fabric. Such a soft-core often results in an ultimately efficient solution to the given application problem, but often lacks flexibility. Motuk et. al. [6],[7] designed FPGA-targeted soft-cores for solving plate equations and obtained impressive efficiency results with this. Using these soft-cores, however, makes it still not trivial to compose a complete instrument model. Pfeifle et. al. has to our knowledge been the first to design FPGA targeted soft-cores that implement complete instrument models, based on the composition of different FDTD models [8]. 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 (i.e. to design soft-cores that enable efficient and versatile model development) and accessible (i.e. without the steep learning curve to apply this technology).

2. SCOPE AND CONTRIBUTION

We focus on the application of FPGA technology for real-time musical instrument modeling in a broad sense. Our special interest is FDTD, but we also take into account that other techniques, like digital waveguide or "classical" analog electronics modeling should also be feasible. Our research object is WaveCore. This is a programmable soft-core processor technology that is specifically designed for audio and acoustics. Earlier work has showed that WaveCore can be applied efficiently for low-latency audio effects processing [9], applying digital waveguide and classical modeling techniques. In this paper we investigate the feasibility of applying the WaveCore technology to real-time physical modeling on FPGA. Our aim is to apply this technology to model complete musical instruments with good efficiency and to hide those technological aspects for the developer that make FPGA difficult to use. The structure of this paper is as follows: we start with an introduction to the WaveCore processor. Then we explain how FDTD schemes can be mapped on WaveCore technology. Further we explain how different FDTD schemes (and possibly other algorithmic parts) can be composed into a larger scope, and conclude with a technology efficiency analysis.

3. DATA-FLOW MODEL AND RELATED PROCESSOR ARCHITECTURE

WaveCore is a many-core processor concept. This processor is a so-called soft-core, which means that the processor design is implemented as a hardware description in a Hardware Description Language (HDL). This HDL code can be targeted (synthesized) to silicon which can be either ASIC (Application Specific Integrated Circuit) or FPGA (Field Programmable Gate Array) technology. After targeting this softcore to the desired technology, the WaveCore processor behaves like a programmable Digital Signal Processor (DSP) chip. The WaveCore processor is programmable with a native language that is inspired by data-flow concepts. This WaveCore programming language is declarative. This means that a WaveCore program describes a function, and hides details about how this function should be executed which is the case in imperative programming languages like C or C++. A WaveCore program is a data-flow process graph: a network of processes, edges and a global scheduler. Each WaveCore Process (WP) is an entity that is periodically executed (i.e. ”fired”) by a global scheduler. Upon firing a process consumes one data-entity, called ”token” from each connected ”inbound-edge”. Likewise, after process execution a process produces one token per associated ”outbound-edge”. A token consists of one or more ”primitive token-elements” (PTE). Where one PTE is the atomic native 32-bit data element. A WP can have an arbitrary number of inbound and outbound edges (including zero). Each WP can have a different link to the global scheduler. This scheduler orchestrates the execution of the entire process graph by periodically generating ”fire-events” to the linked processes through dedicated ”fire-links”. In the example process-graph in fig.1 there are two different fire-links, called Es1 and Es2. This implies that different WP’s can be fired with different rates: a multi-rate process graph is therefore supported by the concept. A global scheduler concept leads to a very strict and predictable execution behaviour of the process-graph, provided that the execution time of the actors is predictable. A WaveCore process is allowed to be partitioned. Such a partition is called a WaveCore Process Partition (WPP). As can be seen in fig.1, an inbound or outbound edge can be divided and linked to different WPP’s. Hence it is possible that multiple WPP’s can contribute to the production/consumption of a single token. At the lowest level we define the ”Primitive Actor” (PA). A PA is an indivisible process which is depicted in fig.2. As can be seen a PA has two inbound edges: $x1, x2$, a function f, one outbound edge y, a parameter p, a max-length delay Λ and an inbound edge τ which controls the actual delay-line length dynamically. Each PA consumes at most two PTE’s, and produces one PTE upon execution of that particular PA. The production of a PTE can be delayed through a run-time controllable delay-line that is associated with each PA. The delay-line length Λ is specified at compile-time and can be run-time modulated with a third inbound signal, called τ, yielding a dynamically variable delay-length $\lambda^{n+1} = \Lambda.g(\tau^n)$. Hence is is possible to describe algorithms with time-varying delays (like flanging [9]) in a compact and efficient way. The maximum delay-length Λ is specified at compile-time and can have any arbitrary positive integer value including zero. A limited set of different PA functions are supported by means of the already mentioned function f (like addition, multiplication, division, etc.). The PA function types that are of importance in this paper are (1) C-type PA: $y^{n+1} = p$, (2) MAD-type PA: $y^{n+\lambda} = p.x_1^n + x_2^n$, (3) ADD-type PA: $y^{n+\lambda} = x_1^n + x_2^n$ (4) MUL-type PA: $y^{n+\lambda} = x_1^n.x_2^n$, and (5) AMP-type PA: $y^{n+\lambda} = p.x_1^n$. The C-type PA is insensitive to its $x1$ and $x2$ inbound edges and basically emits a programmable constant value to its outbound edge, each time that it is fired. This C-type PA can be linked to a token-element and hence to an inbound port or local WPP-input. Each PA can be programmed such that it emits it output PTE to an outbound port, or a partition output, or locally to a fellow-PA within the same WPP.

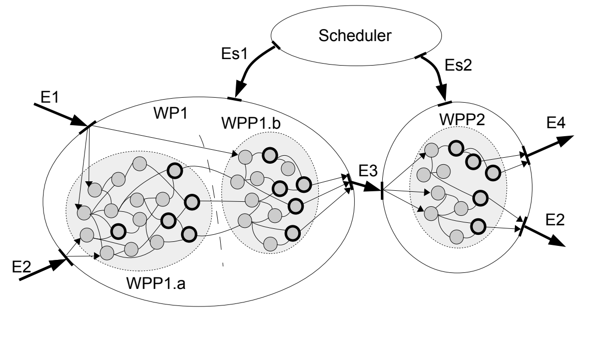

Figure 1: Data-flow oriented WaveCore programming model.

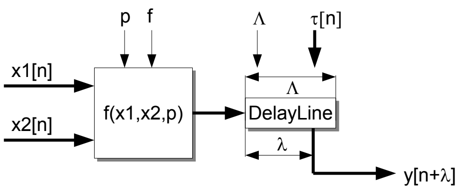

Figure 2: Primitive Actor (PA) model.

3.1. WaveCore processor architecture

The WaveCore processor itself consists of a cluster of so-called ”Processing Units” (PU). A block diagram of this cluster is depicted in fig.3. Each PU embodies a small Reduced Instruction Set Computer (RISC). This PU-cluster is interconnected be means of a Network-on-Chip (NoC), called Graph Partitioning Network (GPN). The PU-cluster can be intialized (i.e. loading compiled WaveCore program code) through the ”Host-Processor Interface” (HPI). This HPI is intended to be connected to a host-computer that can either be on the same chip (FPGA or ASIC) or an externally connected computer. The execution of the individual PU’s within the PU-cluster is orchestrated by the Scheduler. In principle this Scheduler is a programmable sequencer which generates firing events to each individual PU, according to a compile-time derived cyclo-static schedule. Each PU is capable to access system memory space through the ”External Memory Interface”. Hence, each PU is individually responsible for moving token data and delay-line data from/to external memory. The WaveCore compiler (which is not further described in this paper) takes care of automated mapping of a WaveCore Process Graph onto the PU-cluster. Like mentioned each WP in the process graph can be partitioned into one or

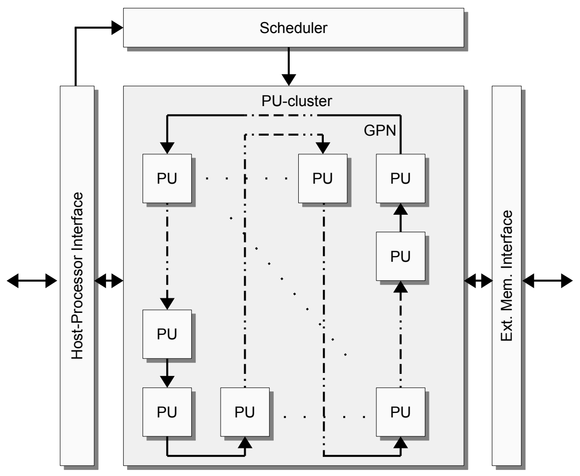

Figure 3: WaveCore processor: PU-cluster.

more WPP's. Each WPP is entirely mapped on a PU. It is supported that one WP is eventually mapped on a set of PU's. This property is used by mapping a finite-difference scheme, as we will explain further in this paper. The connections between the different WPP's are automatically mapped on the GPN network by the compiler.

3.2. Processing Unit architecture

Like mentioned, a PU embodies a small pipelined RISC processor. The block diagram of a PU is depicted in fig.4. This unit consists of two main parts: the "Graph Iteration Unit" (GIU) and the "Load/Store Unit" (LSU). The GIU is a small pipelined CPU which is fully optimized to sequentially fire PA's according to a compile-time derived static schedule. The instruction set for this CPU is defined in such a way that a compiled PA can be executed as a single instruction. This unit fires an entire WPP after that it receives a process-dispatch event from the global scheduler (through a fire-link), and hence executes each compiled PA within this WPP sequentially. The heart of the GIU is a single-precision (32-bits) floating-point ALU. The LSU takes care of external memory traffic. As such, the GIU is decoupled from external memory and hence insensitive to the associated latency.

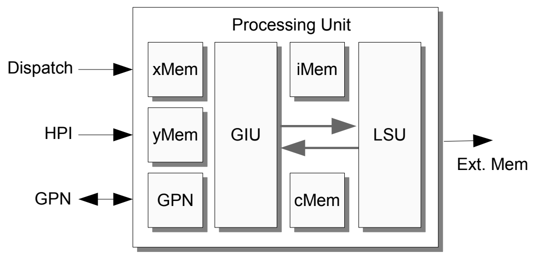

Figure 4: Processing Unit (PU).

4. DESCRIBING PHYSICAL GEOMETRIES IN WAVECORE PROCESSES

In this section we zoom in into the discretization of Partial Differential Equations (PDE) that are of importance for Finite-Difference (FD) modeling [2]. Next, we will analyze and explain how such a scalable and discrete model is translated to a WaveCore process.

4.1. Finite difference schemes of wave equations

We focus on finite difference schemes that originate from the generic wave equation as defined in (1). This equation models wave propagation in a medium, like a string (1-dimensional), plate (2-dimensional) or a 3D medium.

$$\lambda(s).\nabla^k u(s,t) = \epsilon(s)\frac{\partial^2 u(s,t)}{\partial t^2} + \mu(s)\frac{\partial u(s,t)}{\partial t} + f(s,t) \quad (1)$$

Where s represents the spatial coordinates (i.e. $< x >$ for a 1-dimensional geometry or $< x, y >$ for a 2-dimensional geometry). The parameter k represents the order of the differential equation. The first-order partial time derivative term models wave propagation losses. The term $f(s,t)$ enables model stimulation (e.g. plucking a string or striking a snare-drum). In our analysis we will investigate 2nd and 4th PDE orders. We allow the parameters λ, ϵ and μ to be dependent on the spatial coordinates in our analysis. However, we assume these parameters to be time invariant. Physically this means that non-uniformity of the medium is possible. The first step is to transform the space/time continuous eq.(1) into its space/time discrete counterpart. Therefore we introduce a grid in space: $s = \sigma\Delta s$, and time: $t = n\Delta t$. Furthermore we apply a central difference approximation which yields the following substitutions for the first and second partial time derivative:

$$\frac{\partial u}{\partial t} \approx \frac{u_\sigma^{n-1} + u_\sigma^{n+1}}{2\Delta t} \quad (2)$$

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_\sigma^{n-1} - 2u_\sigma^n + u_\sigma^{n+1}}{\Delta t^2} \quad (3)$$

We substitute (2) and (3) into (1) and subsequently solve for u_σ^{n+1}. This yields the iteration function, as defined in (4)

$$u_\sigma^{n+1} = s_1(\sigma)\xi^n(\sigma) + p_1(\sigma)u_\sigma^n + p_2(\sigma)u_\sigma^{n-1} + p_3(\sigma)f_\sigma^n \quad (4)$$

Where we define $\xi^n(\sigma)$ as the spatial stencil computation function. This function accumulates the neighborhood node values, including centered node u_σ^n with discrete timestamp n. The definition of this spatial stencil depends on the dimensions of the models and the order of the differential equation. The general definition is given in equation (5).

$$\xi^n(\sigma) = \sum_k \alpha_k.u_{\sigma_k}^n \quad (5)$$

For a 2D geometry with 2nd order (membrane) we can expand this stencil computation function as follows:

$$\xi^n(i,j) = u_{i-1,j}^n + u_{i+1,j}^n - 4u_{i,j}^n + u_{i,j-1}^n + u_{i,j+1}^n \quad (6)$$

An illustration of a 2D node scheme with the stencil defined in (6) is given in fig.5.

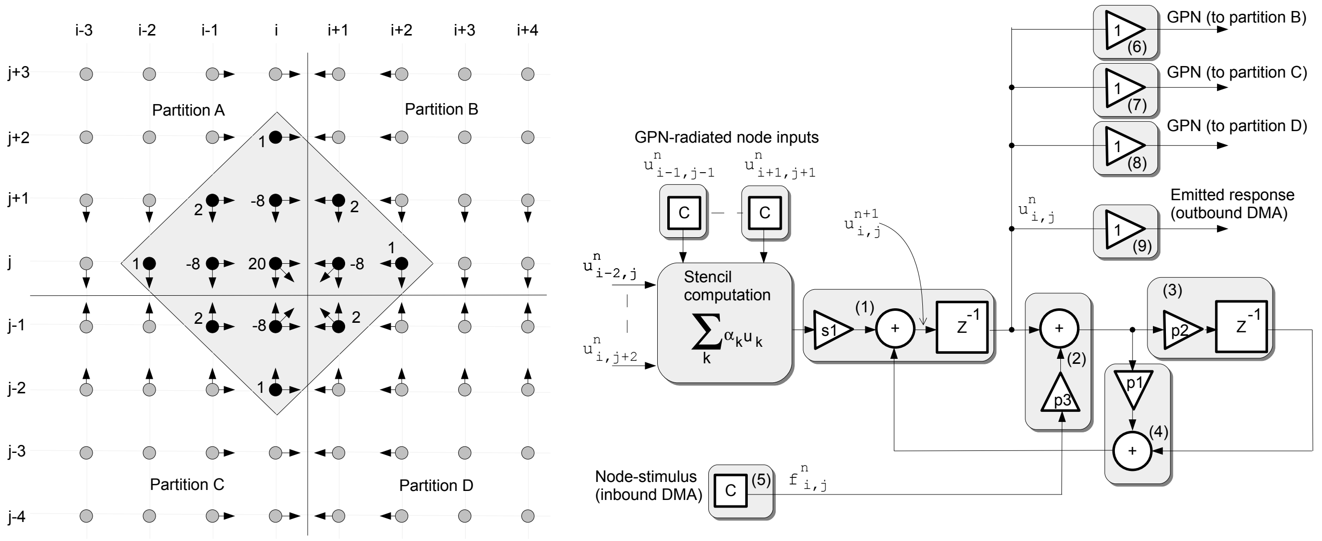

Figure 6: Node computation scheme

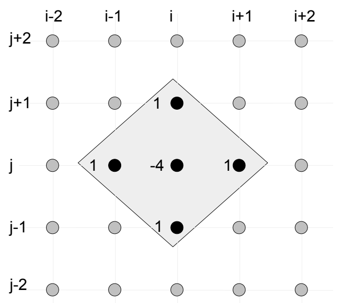

Figure 5: 5-points stencil within membrane model

4.2. Capturing finite-difference schemes into WaveCore processes

A finite-difference scheme as in fig.5, with node-computation defined in eq.(5) can be seen as a directed graph. The basic elements in such a graph are vertices (the nodes wherein the actual computation takes place) and edges. Data elements (i.e. tokens) are carried over the edges and consumed/produced by the vertices. A WaveCore process can also be seen as a directed graph, where the vertices are represented as Primitive Actors (PA), which are arbitrarily interconnected by edges. Hence, a finite-difference scheme can be mapped on a PA-graph. The graph structure of a FD-scheme is very similar to the associated WaveCore PA-graph structure. This is exactly the reason why a WaveCore process transparently represents the mapped FD-scheme. There are basically three main requirements for mapping a finite-difference scheme onto a WaveCore process: (1) scalability, (2) the ability to move stimulus data(f^n_σ) to the model for each node and for each time-stamp n, and (3) the ability to move node-data from the model for each node u^n_σ and for each time-stamp n. A WaveCore PU executes compiled PA's sequentially in time, and the PU clock frequency is limited. This implies that in many cases only a limited part of a FD-scheme (depending on its size) fits on a single PU. Therefore a FD-scheme needs to be partitioned and each partition must be mapped on a PU. Partitioning of the FD-scheme and mapping of partitions on different PU's inherently means that stencil-computation for partition-boundary nodes cannot be done without gathering one or more node-values that belong to the stencil from "neighbor" PU's. The GPN-network within the WaveCore PU-cluster is used for this purpose: partition boundary-node values are moved to neighbor partition(s) over this GPN network for every discrete timestamp. We call this phenomenon "radiation" since every partition radiates its boundary values to its direct environment. Note that the radiation intensity depends on the geometry (1D, 2D, 3D) and spatial stencil complexity (order of the PDE). This is shown for a 2D model of a fourth order differential equation (plate equation) in fig.6. Radiation of node values is illustrated with arrows (e.g. node $u^n_{i,j+2}$ within partition A radiates to partition B). As can be seen, in the worst-case some nodes need to radiate to three neighbor partitions. The example node at position (i,j) depends on 7 radiated node inputs from partitions B,C and D and 6 local node values. Node $u^n_{i,j}$ itself radiates to partitions B,C and D. Note that "radiation" is an inevitable side-effect of partitioning, which has also been described within the context of physical model mapping on FPGA by Motu et.al [7].

The actual mapping of a FD-node onto a PA-graph is depicted in fig.6. The stencil computation is implemented with a multiply-add type PA tree structure (shown as a "black-box"). The inputs of this part of the PA-graph are C-type PA's (if applicable: GPN radiation entry-points) and outputs from different local (i.e. within the same partition) nodes. The output of this stencil computation part feeds into the temporal difference

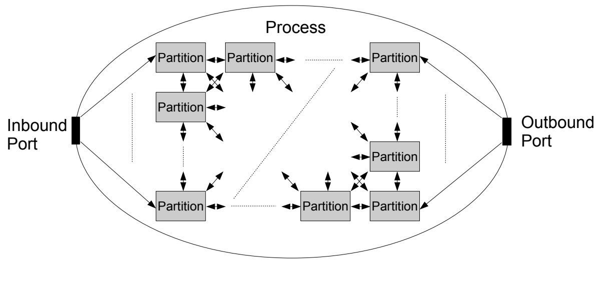

Figure 7: FD scheme represented as a WaveCore process

equation part. This part consists of four interconnected PA's (labeled 1,2,3,4). As can be seen, there is a transparent relationship between the node-PA graph structure and the implemented difference equation (4). The example node (i, j) itself radiates to three neighbor partitions. The PA's 6,7 and 8 are included for this purpose. The execution of each of these PA's (6,7,8) initiate a GPN-write transaction. Moving stimulus data (function f_σ^n) from outside to the model is implemented with C-type PA(5). This PA is linked to the inbound port of the associated WaveCore process. The LSU of the PU takes care that the required data is fetched from external memory through DMA and moved to the memory location that is allocated to PA(5). Moving node data from the model to outside is implemented with PA(9). This PA is linked to the outbound port of the associated WaveCore process. The LSU of the PU takes care that the PA(9) output data is moved to, and the input to PA(5) is moved from external memory space via DMA. The abstract representation of a complete FD-scheme as a WaveCore process (including stimulation and response emission) is depicted in fig.7.

5. COMPOSITION OF PHYSICAL GEOMETRY MODELS IN WAVECORE PROCESS-GRAPHS

The ability to capture finite-difference models in a scalable processor technology is an important requirement to build musical instrument models. This requirement on itself however is not sufficient. Other important requirements are (1) the ability to combine different geometry models where each model may run at a different sampling-rate, (2) the ability to move stimulus and response streaming data between the model and the external (e.g. analog or an externally connected computer that generates stimuli and post-processes generated model data), (3) real-time processing with low latency, (4) the ability to add processes which structures are not as regular as FD-schemes (e.g. traditional signal-processing functions or effects), (4) programming flexibility and processor architecture abstraction. In the following subsections we will demonstrate by three experiments how these requirements fit on the WaveCore programming model and processor technology.

5.1. Experiment 1: primitive six-string device in an acoustic room

We combined a simple guitar model with a virtual guitar player in an acoustic room and described this in a WaveCore process graph that is depicted in fig.8. The virtual guitar player is modeled as a process called "SixStringPlayer". The implementation is a low-frequency oscillator which output if connected to a tapped delay-line. As a result, this process emits a "downstrum" event at the LFO rate (sub-Hz frequency) by periodically generating a "pluck" token. This token consists of six PTE's (one PTE for each string). The guitar model is described in the "Fdtd_SixString" process. This process is composed of six 1D FD-scheme instances (each one coded as a WPP) that are connected to a "SixStringBody" WPP. The six string models are connected to the "pluck" port where each string receives one out of six PTE's from the associated inbound token. Each string is tuned in a way that the guitar model is tuned in an E-major chord. Each pluck event causes a raised-cosine shaped pluck at a fixed position in the associated string. The "SixStringBody" combines the outer edges of the six strings, combines these and routes these back to the same strings (simple interference model). Moreover, this WPP extracts a stereo audio signal from the combined string vibrations and links this stereo signal to the "SixStringOut" port of the process. The "SixStringOut"

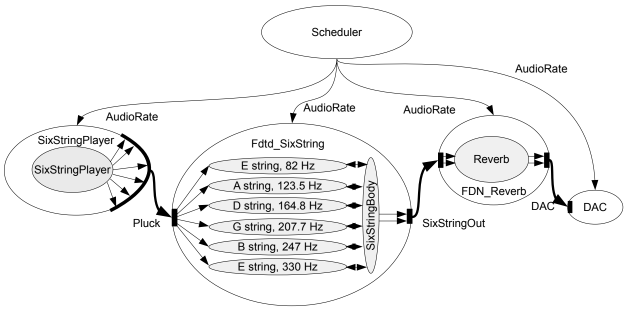

Figure 8: WaveCore Process Graph of six-string model.

port of the "Fdtd_SixString" process is connected to a reverberation process called "FDN_reverb". This process implements a feedback-delay network that models a 3D space in an abstract way (zita_rev1, [10]). This is a simple model in terms of number of PA's, but fairly complex in terms of number of instantiated delay-lines (66 in total). This process yields a reverberated stereo output, called "DAC". Finally the "DAC" edge is connected to a process that is called "DAC". This is not a WaveCore process, but an abstract model of a Digital to Analog Converter. All processes in the graph are synchronized to a single rate called "AudioRate" which is generated by the global scheduler model in the same process graph. The WaveCore compiler translates the entire process graph to an object file. This object file can either be mapped to a target WaveCore processor platform or simulated by the WaveCore simulator. For this process graph we did both and noticed no differences between the simulated model and the real-time (48kHz sampling rate) WaveCore processing on the FPGA board (Digilent Atlys [11]) that we used.

5.2. Experiment 2: scalable 2D plate model

We wrote a 2D plate model generator with 2nd and 4th order differential equations, based on equation (4). This generator produces a scalable and partitioned WaveCore process graph which is depicted in fig.9. The purpose of this process graph

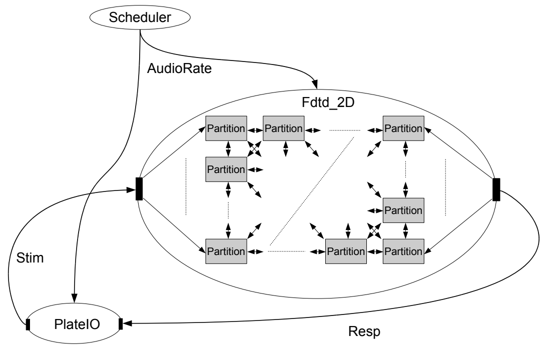

Figure 9: WaveCore Process Graph of 2D plate model with stimulus generation/response capturing.

is to demonstrate the scalability of FD-schemes as partitioned WaveCore processes. Furthermore, we analyzed the mapping characteristics using this model. The results of this analysis are summarized in a next section of this paper. We applied square-shaped partitioning where we tried to optimize the partition size in such a way that a partition just fits on a targeted

PU. The "PlateIO" process enables verification of the model: this process injects stimuli into the model and captures emitted responses. We used the WaveCore simulator to verify the functional correctness of the generated model. A simulation snapshot is depicted in fig.10 We observed that a FD-scheme is

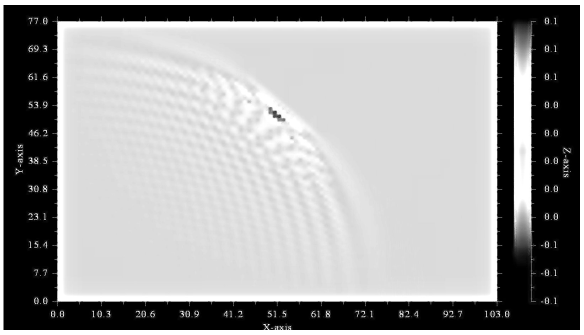

Figure 10: WaveCore simulation snapshot of 2D plate model.

uniformly scalable. The WaveCore compiler turned out to be robust and fast: it took less than a minute to compile an FD-model with over 8000 nodes to WaveCore object code.

5.3. Experiment 3: real-time 2D plate model on FPGA

We generated a 2D membrane model (2nd order PDE) and compiled this as a partitioned WaveCore process which is depicted in fig.11. Within this process graph we connected the analog

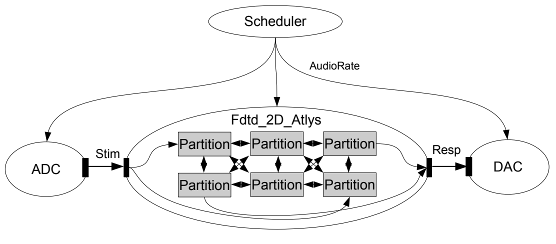

Figure 11: WaveCore Process Graph of 2D plate model on FPGA, connected to ADC/DAC.

stereo input of the FPGA to two opposite diagonal corners of the 2D node-grid. Likewise, we connected the other two diagonal corner node outputs of the 2D node-grid to the stereo output of the FPGA board. These analog I/O connections are modeled in the WaveCore process graph by the ADC and DAC processes. This model runs real-time at 48kHz audio sampling rate and ultra low latency where we connect an audio source to the analog input of the board and a speaker to the analog output of the board. We also simulated this process graph using the WaveCore simulator. We used an audio sound bite that we linked to the ADC process, and played the produced (by the DAC process) audio with a media player. We observed a similar audible result, compared with the FPGA setup.

6. MAPPING EFFICIENCY

The efficiency of a processor technology can roughly be divided into two aspects. The first aspect is the compiler efficiency: for a given application domain, how efficient are the algorithms mapped on the target processor architecture. The second aspect is what we call "silicon efficiency": how efficient is the given processor architecture w.r.t. silicon area. The overall efficiency is ultimately a metric on how efficient the algorithms within the target application domain are mapped on the overall processor technology. For physical modeling on WaveCore we have investigated mapping efficiency, silicon efficiency for FPGA and the overall efficiency.

6.1. Compiler efficiency

The mapping of a FD-scheme on the WaveCore processor is most efficient when all the PU's in the cluster execute arithmetic PA's for every PU instruction. This means that all processing effort is spent on the actual computation. Inefficiency is caused by a non-ideal locality of reference: we introduced the concept "radiation" which implies explicit data movement and replication of node-data over partition boundaries. The compiler itself also generates inefficiency that is caused by a non-optimal memory allocation and pipeline scheduling. This inefficiency results in so-called "bubbles" in the GIU pipeline (caused by data dependency conflicts of subsequent instructions) and additional instructions for operand pointer initialization. We investigated the compiler efficiency for FD-scheme partitions. We optimized these partitions in such a way that the size just fits a PU, where we fixed the PU instruction size to 2048. We investigated mapping capabilities for 1D schemes (2nd order) and 2D schemes (membrane and plate models). For each partition we investigated the mapping influence of model stimulation (stim.: the percentage of nodes that are stimulated for every discrete timestep) and response data movement (resp.: the percentage of node values that are moved from the model for every discrete timestep). Table 1 summarizes the PU-mapping efficiency figures. The "GPN-load" figure expresses the percentage of clock-cycles in which GPN transactions (radiation) are initiated, relative to the overall thread-size. The "Comp. Overh." figures expresses the percentage of added instructions, due to compile inefficiencies. Ultimately, the ILP (Instruction Level Parallelism) number is an important metric which expresses the averaged number of required GIU instructions per FD-node.

Table 1: *FD-partition mapping on WaveCore PU*

Dim/ Spatial	Nodes	GPN-load (%)	Comp. Overh. (%)	Stim (%)	Resp (%)	ILP
1/2	292	0.06	4.51	1	1	5.56
2/2	169	3.22	4.10	1	100	11.10
2/2	169	3.66	4.93	1	1	7.14
2/4	81	4.54	6.39	1	100	20.00
2/4	81	4.72	8.22	1	1	16.67

6.2. Silicon efficiency

We mentioned that the WaveCore processor is implemented as a softcore and described in a HDL (Hardware Description Language). This HDL implementation is configurable at compile-time (within this context "compilation" means the process of mapping the HDL description of the processor to the targeted technology). The most important configuration parameters are PU-cluster size (number of PU's) and PU memory configuration (i.e. instruction memory size per PU). The HDL description of a configured WaveCore processor instance can be compiled to a target technology: either FPGA or ASIC. The target technology, combined with application requirements dictates the optimal processor configuration. Typically the most important constraints within a given technology are the amount of embedded memories, feasible clock frequency and the amount of logic-gates. The execution of a PU-thread is always linear: a PU does not support branches, loops or jumps in the program flow. This means that the number of instructions that can be executed during one sampling period can be expressed by eq.(7).

$$C_p(f_s) = \begin{cases} N_{pu}.C_{imem} & \text{if } f_s \leq f_d \\ N_{pu}.f_{clk}/f_s & \text{if } f_s > f_d \end{cases} \quad (7)$$

With $C_p(f_s)$ the PU-cluster capacity as function of the sampling frequency, N_{pu} the number of PU's in the cluster, f_{clk} the PU clock frequency, f_s the sampling frequency and C_{imem} the instruction memory capacity. This hyperbolic relation between PU capacity and sampling rate dictates the optimization of a PU-cluster for a given application domain and a given technology (e.g. FPGA), with $f_d = f_{clk}/C_{imem}$ as "drop-off" sampling frequency" (the capacity C_p drops for sample frequencies $f_s > f_d$). We used the relation in eg.(7), and the FPGA

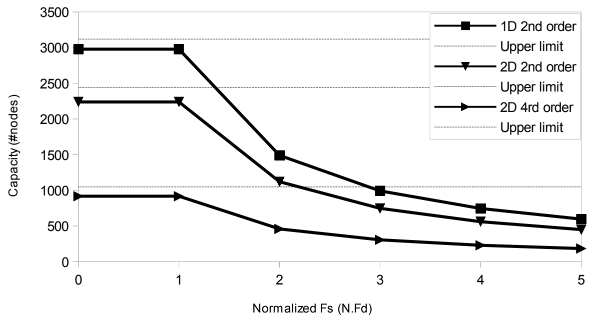

Figure 12: FDTD mapping capacity curves of WaveCore on Atlys FPGA board

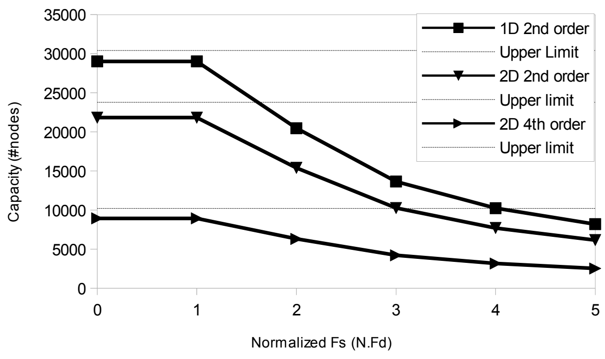

Figure 13: FDTD mapping capacity curves of WaveCore on ML605 FPGA board

constraints to derive PU-clusters for two target platforms: the Digilent Atlys board (Spartan6, LX45 device) and the Xilinx ML605 board (Virtex6). Our goal has been to utilize the available FPGA resources as much as possible. For the Atlys board we derived a PU-cluster which consists of 8 PU's and 2k instruction capacity per PU. For the ML605 board we derived a PU-cluster which consists of 55 PU's where 23 PU's have 4k instruction capacity each and the other 32 PU's have 2k instruction capacity each. The resulting FDTD mapping capacity profiles for both mentioned boards are depicted in fig.12 and fig.13. These capacity profiles represent the capacity (in #nodes) for the scalable 1D and 2D (2nd and 4th order) FD geometries that we analyzed in the previous sections of this paper, as a function of the normalized sampling frequency. The "upper-limits" in the figures represent the theoretically maximum performance, with no compiler efficiency losses. The actual performance depends on the achieved clock-frequency f_{clk}. The maximum f_{clk} depends to a large extend on the GIU pipelining and has an upper-limit that is defined by the floating-point arithmetic. For the Atlys board we found a feasible clock frequency: $86MHz < f_{clk} < 189MHz$, or $42kHz < f_d < 92kHz$. For the ML605 board we found a feasible clock frequency: $150MHz < f_{clk} < 338MHz$, or $37kHz < f_d < 82kHz$. The lowest clock-frequencies are obtained without any pipeline optimizations (push-button HDL compilation). The highest frequencies require careful pipeline optimizations.

6.3. Overall efficiency

Ultimately, the most interesting efficiency characteristics are determined by the ability to map real-life application cases (virtual instruments) onto a given processor technology. Estimation of model complexity is not straightforward since the complexity of a physical model of a musical instrument in terms of #nodes depends on may aspects, such as the frequency-dependent speed of sound on wood and other material and geometry dependent parameters. The required sampling rates for this application domain vary according to the nature of the modeled objects and may range up to approximately 500 kHz. An example of a realistic instrument model is the Banjo. This model is composed of 5 strings (560 nodes in total, 1D, 65kHz sampling rate), a membrane (2048 nodes, 2D circular shaped, 128kHz sampling rate), an air-filled cilindrical shaped 3D space (8192 nodes, 3D, 65kHz sampling rate) and the bridge (512 nodes, 2D, 128kHz sampling rate) which is the interface between the strings and the membrane. Pfeifle et. al. have been able to map a fully functional Banjo model using the methodology as described in [8] on the ML605 board where the majority of the FPGA resources are utilized. We made an estimation on the feasibility to map this model on the ML605 board using WaveCore technology, based on the obtained capacity curves in fig.13 and reasonable estimations of required composition interface overhead. The outcome of this (optimistic) estimation is that it is probably feasible to map an instrument of this complexity on the ML605 board where approx. 75% of the WaveCore cluster capacity would be utilized. However, a couple of performance/efficiency optimizations will need to be done to the technology. (1) The required minimum PU cluster clock frequency will be approximately 250MHz. (2) Automated partitioning of geometry parts to maximally utilize the PU capacity is necessary. (3) We found out that data memory allocation (performed by the WaveCore compiler) is too greedy, resulting in a too large data memory footprint. A more efficient data memory allocation however is conceptually possible.

7. CONCLUSIONS AND FUTURE WORK

We have investigated the feasibility of using WaveCore processor technology for real-time physical modeling purposes. We have focused on the ability to map FD-schemes with different dimensions and different PDE orders. Furthermore we focused on the ability to use this technology to compose virtual musical instrument models. We did this by demonstrating the process-graph oriented programming methodology that enables the combination of different processes that are possibly multi-rate and are possibly implementations of different modeling methods (e.g. FD-schemes, combined with FDN or arbitrary dataflow oriented structures). We observed a high level of transparency between the physical models and the actual implementation of such models in WaveCore language. We also observed a good efficiency, despite the fact that there is a reasonably high versatility in the programming capability and abstraction. In particular, the programming versatility and abstraction combined with a good mapping efficiency results in our conclusion that WaveCore is a promising processor technology for virtual instrument modeling. Future work is the mapping of one or more musical instrument models on a suitable FPGA platform where the biggest challenges seem to be high sampling rates for some geometry parts and high-bandwidth off-chip response data funneling. We have found possible optimization areas within the technology. Automated partitioning of large process-graphs is an important though currently missing step in the mapping process. Next to this we have found a few optimization options that can further enhance the efficiency of the processor technology.

8. REFERENCES

[1] Julius O. Smith, *Physical Audio Signal Processing for virtual musical instruments*, W3K Publishing, 1st edition, 2010.

[2] Stefan Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, Wiley, 1st edition, 2009.

[3] Udo Zölzer, *DAFX: Digital Audio Effects*, Wiley, 2nd edition, 2011.

[4] Stefan Bilbao, Alberto Torin, Paul Graham, James Perry, and Gordon Delap, *Modular Physical Modeling Synthesis Environments on GPU*, 2014.

[5] Bill Hsu and Marc Sosnick-Pérez, "Real-time gpu audio," *Commun. ACM*, vol. 56, no. 6, pp. 54–62, June 2013.

[6] E. Motuk, R. Woods, and S. Bilbao, "Implementation of finite difference schemes for the wave equation on fpga," Philadelphia, United States, 2005, IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 237–240.

[7] E. Motuk, R. Woods, and S. Bilbao, "Fpga-based hardware for physical modelling sound synthesis by finite difference schemes," in *Proceedings of the IEEE field programmable technology conference*, Singapore, 2005, IEEE, pp. 103–110.

[8] F. Pfeifle and R. Bader, "Real-time finite-difference physical models of musical instruments on a field programmable array (fpga)," in *Proceedings of the 15th int. Conference on Digital Audio Effects (DAFx-12)*, York, UK, 2012.

[9] M. Verstraelen, G.J.M. Smit, and J Kuper, "Declaratively programmable ultra low-latency audio effects processing on fpga," in *Proceedings of the 17th int. conference on Digital Audio Effects*, Erlangen, Germany, 2014, DAFx 2014, pp. 263–270.

[10] Julius O. Smith, "Zita-rev1," Available at `https://ccrma.stanford.edu/~jos/pasp/Zita\_Rev1.html`, accessed Jan. 21, 2014.

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