A Minimalistic Architecture for Reconfigurable WFS-Based Immersive-Audio

Dimitris Theodoropoulos
D.Theodoropoulos@tudelft.nl

Georgi Kuzmanov
G.K.Kuzmanov@tudelft.nl

Georgi Gaydadjiev
g.n.gaydadjiev@tudelft.nl

Computer Engineering Laboratory, EEMCS, TU Delft
P.O. Box 5031, 2600 GA Delft, The Netherlands
http://ce.et.tudelft.nl

***Abstract*—We propose a minimalistic processor architecture tailoring Wave Field Synthesis (WFS)-based audio applications to configurable hardware. Eleven high-level instructions provide the required flexibility for embedded WFS customization. We describe the implementation of the proposed instructions and apply them to a multi-core reconfigurable WFS architecture. Our approach combines software programming flexibility with improved hardware performance and low power consumption. Experimental results suggest that our Virtex4FX60-based FPGA prototype, running at 100 MHz, can provide a kernel speedup of up to 4.5 times compared to an OpenMP-annotated software solution implemented on a Core2 Duo at 3.0 GHz. Furthermore, when larger FPGAs are utilized, we estimate that our system can render in real-time up to 32 acoustic sources when driving 64 loudspeakers. Ultimately, we estimated that the proposed system requires approximately 6 Watts, which is at least an order of magnitude less power compared to x86-based approaches.**

I. Introduction

The Wave Field Synthesis (WFS) is a technique that improves substantially the sound quality over stereophony [1]. Moreover, in order to generate the original acoustic wavefronts [2], it requires large loudspeakers arrays that need to be properly driven. Research on literature reveals that the majority of experimental and commercial WFS systems are implemented using General Purpose Processors (GPPs). The primary reason is due to their high-level programming environment, thus a system under development can be tested more rapidly. For example, the designer has the option to easily select different system parameters, such as the number of loudspeakers or the Finite Impulse Response (FIR) filter coefficients sets, in order to conduct various experiments. However, two drawbacks are introduced, namely limited processing capabilities and excessive power consumption.

In order to alleviate these obstacles, we propose a minimalistic processor architecture[1] for embedded WFS. The supporting programming model allows a high-level interaction with a custom-hardware WFS processor, thus alleviating the need of long-time iterations to re-test the system that is under development. Moreover, our proposal combines the programming flexibility of software approaches with the high performance and comparatively lower power consumption of the contemporary reconfigurable hardware. The architecture implementation allows utilization of varying number of processing elements, therefore, it is suitable for mapping on reconfigurable technology. More specifically, the contributions of this paper are the following:

- We propose a unique minimalistic processor architecture, which is specialized for WFS processing and consists of eleven instructions, a dedicated memory organization and a Special Purpose Register (SPR) file. The architecture is scalable and allows programmer's control over the underlying micro-architectural configuration. Thus, once written, the same program can be executed on various implementation configurations.
- We implement a hardware prototype of our architecture as an embedded Multi-Core WFS Processor (MC-WFSP) on a V4FX60 FPGA. Our prototype can render up to 32 real-time sources when driving 56 loudspeakers, while larger FPGAs could accommodate systems that support 64 loudspeakers.
- Experimental results suggest that our prototype can process data 4.5 times faster compared to an OpenMP-annotated software implementation on a Core2 Duo running at 3.0 GHz. Also, our hardware design provides a power-efficient solution. It consumes approximately 6 Watts, which is an order of magnitude less power compared to x86-based systems that require tens of Watts when in operation mode.

The rest of the paper is organized as follows: Section II provides a brief background on the WFS technique and references to systems that utilize it. In Section III, we propose our architecture, while Section IV presents its hardware implementation. In Section V we compare our prototype against a software approach and related work. Finally, Section VI concludes the paper.

II. Background and Related Work

Background: As it was mentioned in Section I, the WFS utilizes loudspeaker arrays to render all acoustic sources. In order to drive the i-th element of an L-sized array with

[1]Throughout this paper, we adopt the terminology from [3], according to which, the computer architecture is termed as the conceptual view and functional behavior of a computer system as seen by its immediate viewer - the programmer. The underlying implementation, termed also as micro-architecture, defines how the control and the datapaths are organized to support the architecture functionality.

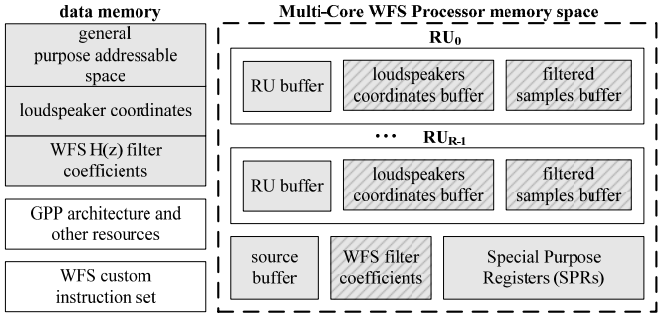

Figure 1. Logical organization of the registers and the memory.

coordinates I(xl_i, yl_i), where *i*=0,..., *L*-1, so as the rendered sound source location is at $A(x_A, y_A)$, the so called *Rayleigh 2.5D operator* [4] has to be calculated:

$$Q_m(\omega, |\overrightarrow{IA_i}|) = S(\omega)\sqrt{\frac{jk}{2\pi}}\sqrt{\frac{Dz}{z+Dz}}\frac{z}{|\overrightarrow{IA_i}|}\frac{exp(-jk|\overrightarrow{IA_i}|)}{\sqrt{|\overrightarrow{IA_i}|}} \quad (1)$$

where *k*=ω/c is the wave number, c is the sound velocity, z is the inner product between a vector $\overrightarrow{n}$ perpendicular to the array and $\overrightarrow{IA_i}$, *Dz* is reference distance, i.e. the distance where the Rayleigh 2.5D operator can give sources with correct amplitude, $S(\omega)$ is the acoustic source, $\sqrt{\frac{jk}{2\pi}}$ is a 3dB/octave correction filter, $\sqrt{\frac{Dz}{z+Dz}}\frac{z}{|\overrightarrow{IA_i}|}$ is the source amplitude decay, and e^{-jkr} is a time delay that has to be applied to the particular loudspeaker. Further details on WFS can be found in [4], [5], [11], and [12].

Related work: References [6] and [7] present two similar audio systems that can be used for teleconferencing scenarios. In both cases, the authors have employed microphone arrays to record audio sources. The latter are sent through Ethernet to a remote location and rendered using a loudspeaker array. Both implementations are based on desktop PCs. Another experimental WFS system is the Binaural Sky [8], which utilizes a Linux PC to synthesize focused sound sources around the listener through a 22-loudspeaker array.

SonicEmotion [9] and Iosono [10] are two companies that produce audio systems based on the WFS technology. SonicEmotion develops its unit on Core2 Duo-based WFS product, which supports rendering up to 64 real-time sound sources, while driving a 24-loudspeaker array. Iosono also follows a standard PC approach that supports up to 32 real-time sources while driving 32 loudspeakers. A cinema in Ilmenau, Germany[2] has been equipped with 192 loudspeakers since 2003, which are driven by six Iosono PCs.

III. WFS Architecture

Memory and registers organization: Figure 1 illustrates the logical organization of the memory and the registers of the proposed WFS architecture. It is assumed that it operates in a co-processor architectural paradigm. The architecture assumes multi-core processing, distributed among *R* Rendering Units (RUs), coupled to the GPP architecture, each processing data for $\frac{L}{R}$ output channels, where *L* is the total number of loudspeakers. The *R* parameter can be determined both at design-time and at run-time. The latter option makes it suitable for implementations on platforms with partial configuration capabilities. The host GPP and the MC-WFSP exchange synchronization parameters and memory addresses via a set of Special Purpose Registers (SPRs). Each RU has its own *RU buffer* and memory space for the loudspeakers coordinates and filtered source samples, namely *loudspeaker coordinates buffer* (LCB) and *filtered samples buffer* (FSB) respectively. Furthermore, there is a *source buffer*, where samples of an acoustic source are stored, and a memory space for the currently active coefficients set of the WFS filter. The non-user addressable space is annotated with the stripe pattern.

WFS instruction set: Table I summarizes the eleven proposed high-level instructions, divided into four categories. The *SPRs modified* column shows which SPR is modified by a particular instruction. Table II provides each SPR functionality. The four different categories of instructions are: *I/O*, *system setup*, *data processing* and *debug*. The *I/O* instruction is used to enable or disable audio streaming to RUs. The *system setup* instructions are used to customize system parameters and reload coefficients to the WFS filter. The *data processing* instruction is used to process loudspeakers audio samples. Finally, the instruction that belongs to the *debug* category, provides an interface to the processor, in order to read any SPR. In the following, we describe each of the instructions and their parameters. We assume that a WFS system consists of *R* RUs and *L* loudspeakers.

OutStreamEn: Enables or disables streaming of audio samples from the RUs to the output channels. Its parameter is a binary mask *b_mask* equal to the number of RUs *R*. Within the mask, each bit can be used from the programmer to disable or enable RU streaming by setting 0 or 1 to its value respectively. The binary mask is stored in SPR0.

ClrSPRs: Clears the contents of all SPRs.

ConfL: Defines the number of loudspeakers that will be processed from a single RU. Its parameter is the number of loudspeakers per RU *spkr_num* that will be enabled using *OutStreamEn*. The instruction writes the value of *spkr_num* to SPR2.

ClrRUBufs: Clears the contents of all *RU buffers* that are currently configured to the system. Its parameter is an array *RUs_addresses* of pointers to each *RU buffer*. No SPR is modified during its execution.

StC2RUs: Reads the loudspeakers coordinates from the external memory, re-arranges their order based on the number of RUs at the system, and writes them to each *RU buffer*. Its parameters are the external memory address *xmem_spkr_coordinates* where the loudspeakers coordinates are stored, and an array of pointers *buf_spkr_coordinates* to the on-chip *RU buffers*. The instruction uses SPR3 to

[2]http://www.idmt.fraunhofer.de/eng/about_us/facts_figures.htm

Table I
PROGRAMMING MODEL FOR WFS APPLICATIONS

Instruction type	Full name	Mnemonic	Parameters	SPRs modified
I/O	Ouput Stream Enable	OutStreamEn	*b_mask*	SPR0
System setup	Clear SPRs	ClrSPRs	NONE	SPR0 - SPR[10+2·R]
	Configure loudspeakers per RU	ConfL	*spkr_num*	SPR2
	Clear RU buffers	ClrRUBuf	** RUs_addresses	NONE
	Store coordinates to RU buffers	StC2RUs	*xmem_spkr_coordinates, **buf_spkr_coordinates	SPR3, SPR11 - SPR[10+R]
	Declare FIR Filter	DFirF	*FSize*	SPR1
	Set Samples Addresses	SSA	**buf_sam_addr	SPR7, SPR[11+R] - SPR[10+2·R]
	Buffer Coefficients	BufCoef	*xmem_coef_addr, *buf_coef_addr	NONE
	Load Coefficients	LdCoef	*buf_coef_addr	SPR4, SPR8
Data processing	Render Source	RenSrc	*Srcx1y1*, *Srcx2y2*, **RUs_addresses, *source_id_num*, *xmem_read_addr, *xmem_write_addr	SPR4, SPR5, SPR9, SPR10
Debug	Read SPR	RdSPR	*SPR_num*	NONE

communicate with the MC-WFSP and writes to SPR11 - SPR[10+R] the address within each *RU buffer* where the arranged coordinates will be stored.

DFirF: Declares the size of the WFS 3dB/octave correction filter. Its parameter is the filter size *FSize*, which is written to SPR1.

SSA: Specifies the addresses from where the MC-WFSP will read the source samples and write the output data. Its parameter is an array of pointers *buf_sam_addr* to the starting address of the *source buffer* and all *RU buffers*. *SSA* writes from SPR[11+R] to SPR[10+2·R] the *RU buffers* starting addresses. Furthermore, it writes to SPR7 the *source buffer* address, where 1024 samples of the source signal are stored.

BufCoef: Fetches all WFS filter coefficients from the external memory to the *source buffer*. Its parameters are a pointer *xmem_coef_addr* to the off-chip memory starting addresses of the coefficients set, and a pointer *buf_coef_addr* within the *source buffer* where all coefficients will be stored. *BufCoef* does not write any values to SPRs.

LdCoef: Loads the WFS filter coefficients to the 3dB/octave correction FIR filter of the system. Its parameter is a pointer *buf_coef_addr* within the *source buffer* where all coefficients are stored. This address is written to SPR8. As soon as all coefficients are transferred to the *source buffer*, their distribution is initiated when a start flag is written to SPR4. Once all coefficients are reloaded to the filter, *LdCoef* writes a done flag to SPR4.

RenSrc: Processes a 1024-sample chunk of streaming source data using every RU that is enabled with the *OutStreamEn* instruction. *RenSrc* requires as parameters the source coordinates (x1,y1) and (x2,y2), which designate the initial and final source location within the listening area respectively for a $\frac{1024}{f_s}$ time interval (f_s is the sampling frequency), and are stored to variables *Srcx1y1* and *Srcx2y2*. Also it requires an array *RUs_addresses* of pointers to each *RU buffer*, the source identification number *source_id_num*, the starting read address from the external memory *xmem_read_addr* of the current chunk, and the write address to the external memory *xmem_write_addr*, where 1024·L output samples will be stored. In order to initiate processing, the instruction writes a start flag to SPR9. This flag is read by each RU, thus loudspeaker processing is performed concurrently. Once all data calculations are finished, a done flag is written to SPR9.

Table II
SPECIAL PURPOSE REGISTERS

SPR	Description
SPR0	OutStreamEn binary mask
SPR1	WFS filter size
SPR2	Loudspeakers per RU
SPR3	StC2RUs start/done flag
SPR4	LdCoef start/done flag
SPR5	x1, y1 source coordinates
SPR6	x2, y2 source coordinates
SPR7	source buffer address
SPR8	WFS filter coefficients address
SPR9	RenSrc start/done flag
SPR10	source ID number
SPR11 - SPR[10+R]	loudspeakers coordinates address inside RU_i buffer, i=0,...,R-1
SPR[11+R] - SPR[10+2·R]	loudspeakers samples address inside RU_i buffer, i=0,...,R-1

RdSPR: Reads any of the SPRs. It requires as parameter the number of SPR *SPR_num* that needs to be read. It does not modify any SPR.

IV. RECONFIGURABLE WFS MICRO-ARCHITECTURE

WFS system description: Figure 2 illustrates in more detail the implementation of the architecture from Figure 1. We should note that it is based on our previous MC-WFSP processor, originally presented in [12], however significant design improvements have been made, in order to support a high-level programming ability.

A *GPP Bus* is used to connect the on-chip GPP memory and external SDRAM with the GPP via a standard bus interface (BUS-IF). Furthermore, in order to accelerate data transfer from the SDRAM to on-chip buffers, a *Direct Memory Access (DMA) controller* is employed, which is also connected to the same bus. A partial reconfiguration controller is used to provide the option of reloading the correct bitstreams based on the currently available RUs. All user-addressable spaces inside the MC-WFSP, like SPRs, *RU buffers* and the *source buffer*, are connected to the *GPP Bus*. This fact enhances our architecture's flexibility, since they are directly accessible by the GPP. The *main controller* is responsible for initiating the coefficients reloading process to the WFS filter and distributing the loudspeakers coordinates to all RUs. Furthermore, it broadcasts all filtered data to each RU, enables output data processing from the selected RUs,

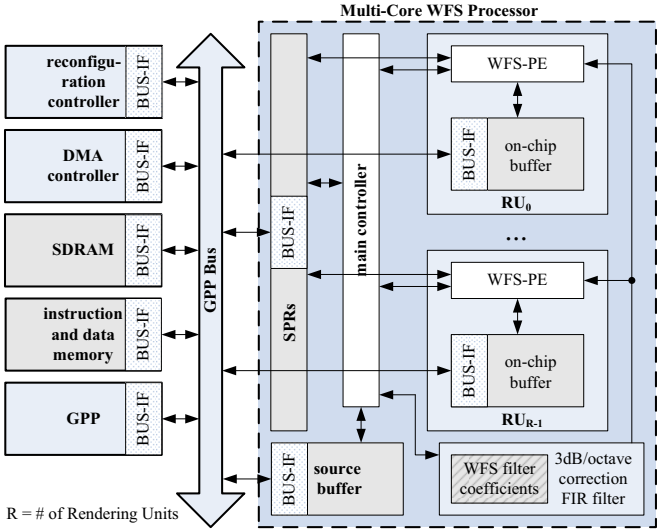

Figure 2. Detailed implementation of the WFS system.

and acknowledges the GPP as soon as all calculations are done.

Within each RU, there is a *WFS Processing Element* (WFS-PE) module, illustrated in Figure 3. The *StCoord controller* is connected to the *main controller* and is responsible for transferring the loudspeakers coordinates from the *RU buffer* to the internal *LCB*. The *Render controller* reads the coordinates of each loudspeaker from the *LCB* and forwards them to the *Preprocessor*. The latter reads the loudspeaker coordinates and calculates the amplitude decay, source velocity and source distance from a particular loudspeaker based on the source current position inside the listening area. The *WFS Engine* module integrates the *FSB* to store all filtered samples. Furthermore, it employs two cores that select the proper samples, based on the source distance from the same loudspeaker, and multiply them with the amplitude decay. All output samples are written back to the *RU buffer*. Further details on the hardware specifications of the *Preprocessor* and *WFS Engine* modules can be found in [12].

Instructions implementation: The instructions are divided into four different categories based on whether they access an SPR or not, and are described in details below:

GPP reads from SPR: *RdSPR* is the only instruction that belongs to this category. The GPP initiates a *GPP Bus* read-transaction and, based on the *SPR_num* value, it calculates the proper SPR memory address.

GPP writes to SPR: *OutStreamEn*, *ClrSPRs*, *ConfL*, *ClrRUBufs*, *DFirF*, and *SSA* are the instructions that belong to this category. When the *OutStreamEn* instruction has to be executed, the GPP initiates a bus transaction and writes the binary mask *b_mask* to SPR0. Similarly, in *ClrSPRs* the GPP performs consecutive bus transactions to access all SPRs and writes the zero value to them. The *ConfL* instruction writes the *L/R* parameter to SPR2 and also forwards the *R* parameter to the partial reconfiguration controller, in order to

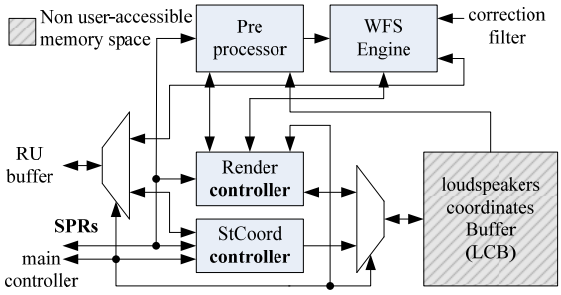

Figure 3. The WFS-PE structure.

load from the external memory the bitstream that includes *R* RUs. The *ClrRUBufs* instruction performs DMA transactions to initialize all available *RU buffers* with zeros. The *DFirF* also performs a bus transaction to write to SPR1 the WFS filter size. Finally, the *SSA* instruction accesses the *GPP Bus* to write the *source buffer* address to SPR7 and all *RU buffer* addresses to SPR[11+R] - SPR[10+2·R].

GPP reads and writes to SPR: *StC2RUs*, *LdCoef* and *RenSrc* instructions belong to this category. When the *StC2RUs* instruction is executed, the GPP performs a DMA transaction to read all loudspeakers coordinates from the external memory address *xmem_spkr_coordinates* and store them the GPP on-chip memory. The loudspeakers coordinates are re-arranged based on the number of RUs of the system and stored to the *RU buffers*. The GPP writes a start flag to SPR3 and remains blocked until a done flag is written to the same register. The start flag is read by the MC-WFSP and the *main controller* invokes the *StCoord controller* to load the coordinates from the *RU buffers* to the internal LCBs. As soon as all loudspeakers coordinates have been transferred, the MC-WFSP writes the done flag to SPR3, which is read by the GPP to continue further processing.

When the *LdCoef* instruction is executed, the GPP performs a bus transaction to write to SPR8 the WFS coordinates address inside the *source buffer*. Furthermore, it writes to SPR4 the start flag and remains blocked until a done flag is written to the same SPR. The MC-WFSP reads the start flag and the *main controller* starts the coefficients reloading to the FIR filter. Once all coefficients are loaded, the MC-WFSP writes a done flag to SPR4, which is read by the GPP to continue further processing.

Finally, when the *RenSrc* is executed, the GPP writes to SPR5 and SPR6 the (x1,y1) and (x2,y2) source coordinates. Furthermore, it writes to SPR10 the source identification number and performs a DMA transaction to read a 1024-sample chunk from the external memory and store it to the *source buffer*. The GPP then writes to SPR9 a start flag and remains blocked until a done flag is written to the same register from the MC-WFSP. The latter reads the start flag and the *main controller* invokes the *render controller* within each RU to start data processing. For every loudspeaker that is processed within a specific RU, the *render controller* reads its coordinates from the LCB and forwards them to the *Pre-*

Table IV
MEASURED AND ESTIMATED EXECUTION TIMES WHEN 32 SOURCES ARE RENDERED USING 4 RUS.

Columns →	1	2	3	4	5	6	7	8	9	10	11	12
Rows	**# of**	**(msec)**	**(msec)**		**(msec)**		**(msec)**		**(msec)**		**Kernel speedup**	
↓	**loudspeakers**	$t_{Core2Duo}$	$t_{PPC-SDRAM}$	$t^{opt}_{PPC-SDRAM}$	$t_{MC-WFSP}$	$t^{opt}_{MC-WFSP}$	t_{PPC-SW}	t^{opt}_{PPC-SW}	t_{HW}	t^{opt}_{HW}	Meas.	Est.
1	16	6854	8752	2917	1495	997	370	248	10617	4162	4.58	6.87
2	32	9398	14978	4993	2481	1654	367	246	**17826**	6893	3.78	5.68
3	48	**12050**	21329	7110	3468	2312	370	248	**25167**	9670	3.47	5.21
4	56	**13391**	24467	8156	3967	2645	370	248	**28804**	11048	3.37	5.06

Table III
RESOURCE UTILIZATION OF EACH MODULE

Module	**Slices**	**DSP Slices**	**Memory(bytes)**
Single RU (100 MHz)	3566	26	36864
Common modules among all RUs	6734	0	2048
MC-WFSP with 4 RUs	20998	104	149504
System infrastructure	3213	0	227328
Complete system with 4 RUs	24211	104	376832

processor to calculate the amplitude decay, source velocity and source distance from the particular loudspeaker. Once these parameters are computed, the *render controller* invokes the *WFS Engine*, which selects the proper audio samples, multiply them by the amplitude decay and store them back to the *RU buffer*. As soon as all assigned loudspeakers to all RUs are processed, the *main controller* writes a done flag to SPR9, which is read by the GPP to continue further processing.

GPP does not access any SPR: *BufCoef* is the only instruction that belongs to this category. The GPP reads all source and destination addresses from the *xmem_coef_addr* and *buf_coef_addr* pointers respectively. Then it performs a DMA transaction to transfer all WFS filter coefficients to the *source buffer*.

V. FPGA PROTOTYPE AND EVALUATION

FPGA prototype: Based on the above study, we used the Xilinx ISE 9.2 and EDK 9.2 CAD tools to develop a VHDL hardware prototype and map it on a Xilinx ML410 board. As host GPP processor, we used one of the two integrated PowerPC processors of the V4FX60 FPGA. Furthermore, we used the Processor Local Bus (PLB) to connect all peripherals, which are all *RU buffers*, the *source buffer*, all SPRs, and the DMA and SDRAM controllers. For the partial reconfiguration, we have used the Xilinx Internal Communication Access Port (ICAP), which is also connected to the PLB. The PowerPC runs at 200 MHz, while the rest of the system is clocked at 100 MHz. Our prototype is configured with *R*=4 RUs.

The first line in Table III provides the required resources for a single RU. The second line indicates all required resources to implement the common modules among all RUs. The third line contains all hardware resources occupied by the MC-WFSP. In the fourth line, we provide the resources required to implement the PLB, DMA, ICAP and all memory controllers with their corresponding BRAMs. The fifth line summarizes all required resources from the entire WFS system. As it can be observed, a single RU requires 3566 slices, which makes it feasible to integrate more such modules within a single chip. According to the data provided in Table III, it is possible for even a medium-sized FPGA to host complete WFS systems with one RU. Moreover, we estimated that larger chips, such as the V4FX100 and V4FX140, could accommodate up to six and seven RUs respectively.

Performance evaluation: We conducted a performance comparison of our hardware prototype against an OpenMP optimized software implementation on a Core2 Duo processor at 3.0 GHz. In both cases, we used audio data with duration of 11264 msec, which implies that each real-time implementation must finish all required calculations within this time frame.

We divided the hardware execution time into the time spent on accessing the external memory ($t_{PPC-SDRAM}$), the actual hardware processing time ($t_{MC-WFSP}$) and the elapsed time when the PPC is running the software (t_{PPC-SW}). We performed experiments and measured these times for 32 sources to be rendered under 16-, 32-, 48- and 56-loudspeaker setups, shown in columns 3, 5, and 7 respectively of Table IV. Columns 2 and 9 present the total Core2 Duo and hardware execution times, $t_{Core2Duo}$ and t_{HW} respectively. The numbers in bold represent the systems that fail the aforementioned real-time constraint.

In column 11 of Table IV, we provide the measured WFS kernel speedup that our FPGA-based system achieved against the Core2 Duo approach. As we can see, the MC-WFSP can process data up to 4.5 times faster. Furthermore, t_{PPC-SW} occupies only a small portion of the overall execution time. In contrast, $t_{PPC-SDRAM}$ dominates the total hardware execution time. This timing analysis leads to the conclusion that the existing memory subsystem introduces a performance bottleneck. Therefore, a faster external memory interface is required, in order to to balance the overall system.

Since the PowerPC405 can operate up to 450 MHz [14] and the PLB v4.6 at 200 MHz with 128 bits width [13], we considered an optimized system with a 128-bit wide PLB at 150 MHz, connected to a 300-MHz PowerPC. We estimated the optimized application execution time t^{opt}_{HW}, which was divided into the external memory access time $t^{opt}_{PPC-SDRAM}$, hardware execution time $t^{opt}_{MC-WFSP}$ and PPC software execution time t^{opt}_{PPC-SW}. The latter are provided in columns 4, 6, 8 and 10 of Table IV respectively. As it can be observed from column 12, the optimized system achieves a kernel speedup up to 6.87 times. Moreover, based on column 10, it can render in real-time up to 32 sources when driving 56 loudspeakers. In contrast, based on column 2, the software approach supports up to 32 sources to be rendered using less than 48 loudspeakers.

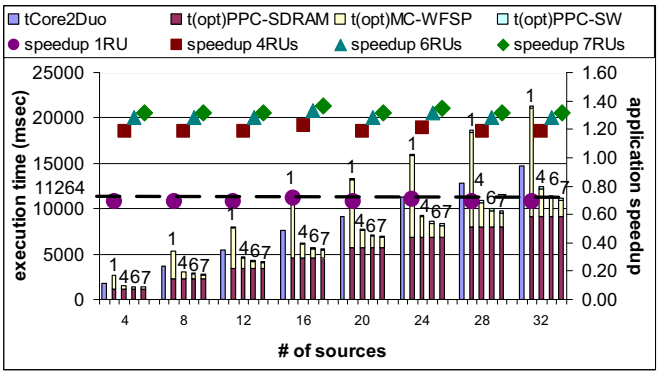

Figure 4. Estimated execution time and application speedup against the software implementation for 64 loudspeakers.

Figure 4 shows the estimated execution time comparison against the Core2 Duo implementation and the achieved speedup, when utilizing a varying number of RUs. The number above each column indicates the number of RUs that each hardware system utilizes. As it can be observed, when there are 32 sources to be rendered from 7 RUs, the $t^{opt}_{PPC-SDRAM}$ occupies 82% of the total hardware execution time. Even though, by applying the aforementioned improvements, the MC-WFSP with 7 RUs can render up to 32 sources through 64 loudspeakers, while the Core2 Duo implementation fails to meet the timing constraints.

Power consumption: Based on the Xilinx XPower utility, our prototype requires approximately 6 Watts. This is an order of magnitude less compared to the Core2 Duo, which requires up to 65 Watts when in operational mode [15]. The power benefits of our approach can be even more substantial in cases where larger WFS systems need to be build. For example, as it was mentioned in Section II, the six rendering PCs that drive the 192 loudspeakers of a cinema in Ilmenau, Germany, and consume hundreds of Watts power, could be replaced by four V4FX60 FPGAs that accommodate our MC-WFSP, which would consume an order of magnitude less power.

Comparison to related work: As it can be observed from Table V, due to the high-level developing environment of the x86, all systems are based on standard PCs. Our FPGA-based solution provides a similarly versatile programming interface, but combines it with high-performance and low power consumption. As we can see from Table V, the MC-WFSP can perform calculations faster or equally well compared to commercial products, and consume an order of magnitude less power than any other x86-based system. Finally, we should note that newer FPGAs, such as the Virtex6 families, provide the potential to increase performance by fitting more RUs functioning at higher frequencies, while reducing power consumption up to 30% [16].

VI. Conclusions

In this paper, we proposed a highly customized processor architecture consisting of eleven instructions for WFS-based audio applications. We described the implementation of the instructions in the context of a multi-core reconfigurable WFS architecture. Our approach combines software programming flexibility with improved hardware performance and low power consumption. Experimental results suggested that the presented MC-WFSP can provide a kernel speedup of up to 4.5 times compared to Core2 Duo-based solutions. Ultimately, larger FPGAs can be utilized to implement more complex WFS audio systems, which at the same time, would require an order of magnitude less power compared to x86-based software solutions.

Table V
Related work summary for WFS implementations.

Reference	# of loudspeakers	Platform	Sources
[8]	22	x86	N/A
[9]	24	x86	64
[10]	32	x86	32
[6]	10	x86	N/A
[7]	24	x86	N/A
MC-WFSP 4 RUs	56	FPGA	32
MC-WFSP 7 RUs	64	FPGA	32

Acknowledgment

This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), FP7 Reflect (grant 248976) and FP6 hArtes (IST-035143).

References

[1] A. Berkhout, et al., "Acoustic Control by Wave Field Synthesis," in *Journal of the Acoustical Society of America*, vol. 93, May 1993, pp. 2764–2778.
[2] G. Theile and H. Wittek, "Wave field synthesis: A promising spatial audio rendering concept," in *Acoustical Science and Technology*, 2004, pp. 393–399.
[3] Gerrit Blaauw and Frederick Brooks, "Computer Architecture: Concepts and Evolution," February 1997.
[4] J. van Dorp Schuitman, "The Rayleigh 2.5D Operator Explained," Laboratory of Acoustical Imaging and Sound Control, TU Delft, The Netherlands, Tech. Rep., June 2007.
[5] W. P. J. D. Bruijn, "Application of Wave Field Synthesis in Videoconferencing," Ph.D. dissertation, TU Delft, The Netherlands, October 2004.
[6] J. Beracoechea, et al., "On building Immersive Audio Applications Using Robust Adaptive Beamforming and Joint Audio-Video Source Localization," in *EURASIP Journal on Applied Signal Processing*, June 2006, pp. 1–12.
[7] H. Teutsch, et al., "An Integrated Real-Time System For Immersive Audio Applications," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, October 2003, pp. 67–70.
[8] D. Menzel, et al., "The Binaural Sky: A Virtual Headphone for Binaural Room Synthesis," in *International Tonmeister Symposium*, October 2005.
[9] SonicEmotion Company, "http://www.sonicemotion.com."
[10] Iosono Company, "http://www.iosono-sound.com."
[11] M. Boone, et al., "Spatial Sound Field Reproduction by Wave Field Synthesis," in *Journal of the Audio Engineering Society*, vol. 43, December 1995, pp. 1003–1012.
[12] D. Theodoropoulos, et al., "Reconfigurable Accelerator for WFS-Based 3D-Audio," in *IEEE Reconfigurable Architecture Workshop*, May 2009.
[13] Xilinx Inc, "Processor Local Bus (PLB) v4.6," December 2009.
[14] ——, "Virtex-4 Family Overview," January 2007.
[15] Intel Corporation, "http://ark.intel.com/Product.aspx?id=33910."
[16] Xilinx Inc, "Power Consumption at 40 and 45 nm," April 2009.