Minimalistic Architecture for Reconfigurable Audio Beamforming

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

***Abstract*—In this paper, we propose a minimal programming model that is tailored to audio Beamforming applications. The model consists of nine instructions that provide high flexibility to customize multi-core reconfigurable beamformers. We describe all instructions and demonstrate their functionality through pseudocode examples. We apply the proposed programming paradigm to a multi-core reconfigurable Beamforming architecture. Our approach combines software programming flexibility with improved hardware performance. Experimental results suggest that our Virtex4FX60-based solution at 100 MHz, can extract in real-time up to 12 acoustic sources 2.6x faster than a 3.0 GHz Core2 Duo OpenMP-based implementation.**

I. Introduction

The Beamforming technique is used in telecommunications to strengthen incoming signals from a particular location. Over the last decades, it has been adopted by audio engineers to develop systems that can extract acoustic sources. Experimental Beamforming systems based on standard PCs provide a high-level programming environment, but they lack performance. Custom-hardware solutions alleviate this drawback, however, in the majority of cases, designers are primarily focused on just performing all required calculations faster than a General Purpose Processor (GPP). Such approaches do not provide a high-level programming environment for exposing various system parameters to the system programmer, like different filter sizes and coefficients sets.

In order to combine a high-level programming environment with improved performance, in this paper, we propose a minimalistic processor architecture[1] for embedded Beamforming. The supporting programming model allows a high-level interaction with a custom-hardware Beamforming processor, thus alleviating the need of long-time iterations to re-test the system that is under development. More specifically, the contributions of this paper are the following:

- We propose a unique minimalistic processor architecture, which is specialized for Beamforming processing and consists of nine instructions, a dedicated memory organization and a Special Purpose Registers (SPRs) file;
- The architecture is scalable and allows programmers' control over the underlying micro-architectural configuration. Thus, once written, the same program can be executed on various hardware implementations;
- A Virtex4FX60 FPGA-based hardware prototype of our embedded Multi-Core BeamForming Processor (MC-BFP), able to extract 12 acoustic sources simultaneously 2.6x faster than a software implementation running on a 3.0 GHz Core2 Duo.

The rest of the paper is organized as follows: Section II provides a brief background on the Beamforming technique and references to various systems that utilize it. In Section III, we propose our architecture, while Section IV evaluates it by demonstrating our instructions through pseudocode. Finally, Section V concludes the paper.

II. Background and Related Work

Background: Beamforming performs spatial filtering over a certain recording area, in order to determine the direction of arrival of incoming signals. Although there are various approaches to perform acoustic Beamforming [2], many systems utilize a filter-and-sum approach, as illustrated in Figure 1. A microphones array of C elements samples the propagating wavefronts and each microphone is connected to an $H_i(z)$, FIR filter. All filtered signals are accumulated, in order to strengthen the extracted audio source and attenuate any ambient noise. Essentially, the FIR filters are utilized as delay lines that compensate for the introduced delay of the wavefront arrival at all microphones [3]. Thus, the combination of all filtered signals will amplify the original audio source, while all interfering ones will be suppressed.

However, in order to extract a moving acoustic source, it is mandatory to reconfigure all filters coefficients according to the source changing location. For example, as it is illustrated in Figure 1, a moving source is recorded for a certain time inside the aperture defined by the $\theta_2 - \theta_1$ angle. A source tracking device is used to follow the source trajectory. Based on its coordinates, all filters are configured with the proper coefficients set. As soon as the moving source crosses to the aperture defined by the $\theta_3 - \theta_2$ angle, the source tracking device will

[1]Throughout this paper, we adopt the terminology from [1], 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.

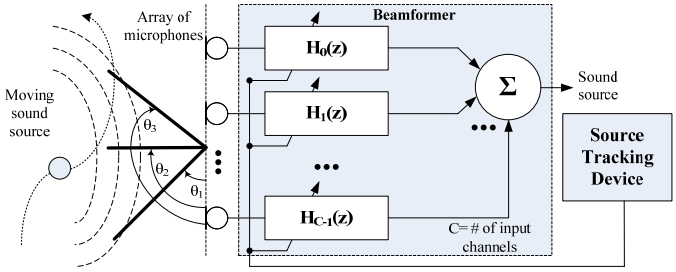

Fig. 1. A filter-and-sum beamformer.

provide the new coordinates, thus all filter coefficients must be updated with a new set. This process is normally referred to as "*beamsteering*".

Related work: The authors of [4] present a Beamforming hardware accelerator, where up to seven instances of the proposed design can fit in a Virtex4 SX55 FPGA, resulting in a 41.7x speedup compared to the software implementation. A DSP implementation of an adaptive subband Beamforming algorithm that utilizes two microphones, is presented in [5]. Finally, an experimental teleconference system is presented in [6], which is based on a standard PC and consists of 12 microphones.

III. Beamforming Architecture

Our proposed architecture comprises dedicated register organization, a specialized instruction set and distributed memory buffers. Furthermore, it targets a multi-core processing paradigm. This allows the design of scalable micro-architectures, with respect to the available hardware resources, which makes the architecture suitable for reconfigurable implementations.

Memory and registers organization: Figure 2 illustrates the logical organization of the memory and the registers of the proposed Beamforming architecture. It is assumed that it operates as an architectural extension of a GPP in a co-processor paradigm. The architecture assumes multi-core processing, distributed among *C* processing modules that process data from *C* input channels. The *C* 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 Multi-Core BeamForming Processor (MC-BFP) exchange synchronization parameters and memory addresses via a set of Special Purpose Registers (SPRs). Each *BeamFormer* module has an on-chip buffer and memory space for the decimator and H(z) filters coefficients. Furthermore, there is also an on-chip *source buffer*, where samples of an extracted source are stored, and a memory space for the currently active coefficients set of the interpolator.

The proposed memory organization of our architecture is the user-accessible memory space illustrated in Figure 2. The non-user addressable space is annotated with the stripe pattern. In order to provide high-level programming environment, the programmer is granted read and write access to the on-chip buffers, the *source buffer*, the external memory and the GPP

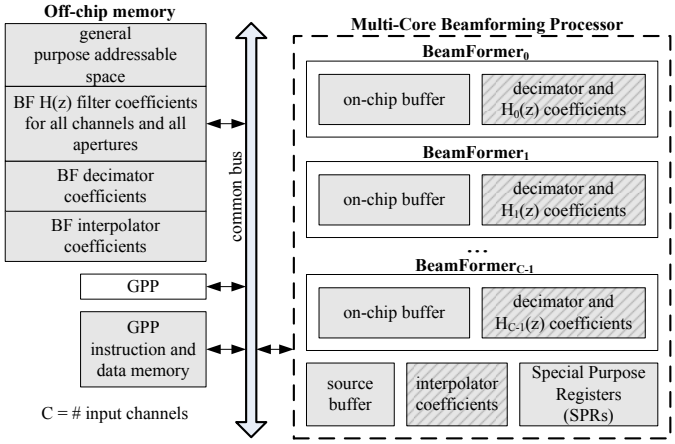

Fig. 2. Logical organization of the registers and the memory.

on-chip memory. Furthermore, the programmer can only read from the SPRs for debugging purposes. There is no direct access to the memory space for the coefficients, since our architecture provides the functionality to reload all required coefficients from on-chip buffers to the decimators, H(z) filters and interpolator. This way the user is shielded completely from all low-level interactions with the hardware.

Beamforming instruction set: The design of a Beamformer requires various tests before its final implementation. Based on the size of the recording area and the hardware costs limitations, the designer has to evaluate the signal-to-noise ratio quality of the extracted sources under different number of microphones. Furthermore, internal signal calculations, except filtering, also require decimation and interpolation. Based on the available hardware resources, the designer should carefully evaluate the size and the filtering coefficients of each one of these modules. In addition, many tests should be conducted to decide the number of source apertures that the recording area should be divided into. Such tests, when developing a software Beamformer, are easily applicable, however this is not the case when custom hardware solutions are required. In the latter case, the designers should also be able to perform easily experiments under different source apertures.

Taking into account these requirements, we provide nine instructions, shown in Table I, for configuring and controlling the MC-BFP from the host GPP software, divided into four categories. The *SPRs modified* column shows which SPR is modified by a particular instruction. 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 processing units. The *system setup* instructions are used to customize system parameters and load filter coefficients to on-chip buffers. The *data processing* instruction is used to process input 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 Beamforming system consists of *C* input channels.

InStreamEn: Enables or disables streaming of audio samples

TABLE I
PROGRAMMING MODEL FOR BEAMFORMING APPLICATIONS

Instruction type	Full name	Mnemonic	Parameters	SPRs modified
I/O	Input Stream Enable	InStreamEn	*b_mask*	SPR0
System setup	Clear SPRs	ClrSPRs	NONE	SPR0 - SPR[9+2·C]
	Declare FIR Filter	DFirF	*FSize*, *FType*	SPR1, SPR2, SPR3
	Set Samples Addresses	SSA	**buf_sam_addr	SPR7, SPR[10+C] - SPR[9+2·C]
	Buffer Coefficients	BufCoef	**xmem_coef_addr, **buf_coef_addr	NONE
	Load Coefficients	LdCoef	**buf_coef_addr	SPR4, SPR8, SPR10 - SPR[9+C]
	Configure # of input channels	ConfC	C	SPR9
Data processing	Beamform Source	BFSrc	*aper*, *xmem_read_addr, *xmem_write_addr	SPR5, SPR6
Debug	Read SPR	RdSPR	*SPR_num*	NONE

from input channels to the Beamforming processing units. Its parameter is a binary mask *b_mask* equal to the number of input channels *C*. Within the mask, each bit can be used from the programmer to disable or enable a single streaming channel by setting 0 or 1 to its value respectively. The binary mask is stored in SPR0 as shown in Table II.

ClrSPRs: Clears the contents of all SPRs.

DFirF: Declares the size of a filter and writes it to the corresponding SPR. Its parameters are the filter size *FSize* and its type *FType*. The latter is used to distinguish among the three different filter types, which are decimator (*FType* = 1), interpolator (*FType* = 2) and H(z) filter (*FType* = 3). Based on the value of *FType*, this instruction writes the filter size to the appropriate SPR ranging from SPR1 to SPR3, as shown in Table II.

SSA: Specifies the addresses from where the MC-BFP will read the input samples. Its parameter is an array of pointers *buf_sam_addr* to the starting address of all on-chip buffers. *SSA* writes from SPR[10+C] to SPR[9+2·C] the on-chip buffers starting addresses. Furthermore, it writes to SPR7 the *source buffer* address, where 1024 samples of the extracted source signal are stored.

BufCoef: Fetches all decimator, H(z) and interpolator coefficients from external memory to on-chip buffers. Its parameters are organized as an array *xmem_coef_addr* of pointers to the off-chip memory starting addresses of the coefficients sets, and an array *buf_coef_addr* of pointers within the on-chip buffers where all coefficients will be stored. *BufCoef* does not write any values to SPRs.

LdCoef: Distributes all decimator and interpolator coefficients to the corresponding filters in the system. Its parameter is an array *buf_coef_addr* of pointers within the on-chip buffers where all coefficients are stored. These addresses are written from SPR10 to SPR[9+C], as explained in Table II. The instruction also writes to SPR8 the on-chip address of the interpolator coefficients from where the MC-BFP can read them. The coefficients distribution is initiated when a start flag is written to SPR4 by the host GPP. Once all filter coefficients are transferred, *LdCoef* writes a done flag to SPR4.

ConfC: Defines the number of input channels that are available to the system. Its parameter is the number of active input channels *C* that will be enabled using *InStreamEn*. The instruction writes the value of *C* to SPR9.

BFSrc: Processes a 1024-sample chunk of streaming data from each input channel that is enabled with the *InStreamEn* instruction, in order to extract an audio source. *BFSrc* requires as parameters the current source aperture *aper*, 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 samples of the source signal will be stored. Based on *aper*, *BFSrc* writes to SPR5 an on-chip buffer address offset that allows the correct selection of $H_i(z)$ coefficients sets. In order to initiate processing, the instruction writes a start flag to SPR6. This flag is read by each $BeamFormer_i$ module, where i=0,..., *C*-1, thus channel processing is performed concurrently. Once all data calculations are finished, a done flag is written by the MC-BFP to SPR6.

RdSPR: Used for debugging purposes and allows the programmer to read any of the SPRs. *RdSPR* requires as parameter the number of SPR *SPR_num* that needs to be read.

TABLE II
SPECIAL PURPOSE REGISTERS

SPR	Description
SPR0	InStreamEn binary mask
SPR1	Decimators FIR filter size
SPR2	Interpolators FIR filter size
SPR3	H(z) FIR filter size
SPR4	LdCoef start/done flag
SPR5	aperture address offset
SPR6	BFSrc start/done flag
SPR7	source buffer address
SPR8	interpolator coefficients address
SPR9	number of input channels (C)
SPR10 - SPR[9+C]	channel i coefficients buffer address, i=0...*C*-1
SPR[10+C] - SPR[9+2·C]	channel i 1024 samples buffer address, i=0...*C*-1

IV. ARCHITECTURE EVALUATION

In Algorithm 1, we illustrate through pseudocode how to setup a Beamforming system to extract an audio source. The *DISABLE_INPUTS_MASK* and *ENABLE_INPUTS_MASK* are binary masks that are used to disable or enable input channels, as described above. The *DECIMATOR_SIZE*, *H_SIZE* and *INTERPOLATOR_SIZE* variables are used to configure the decimator, H(z) and interpolator FIR filter sizes. Moreover, the *DECIMATOR_TYPE*, *H_TYPE* and *INTERPOLATOR_TYPE* variables are used to specify the filter type. *SamplesAddr* is an array of pointers to each on-chip buffer, where a 1024-sample chunk is stored. *CoefXMemAddr* is an array of pointers to the external memory where all required decimator, H(z) filters and interpolators coefficients are stored, and *BufAddr* is an array of destination pointers to on-chip buffers, where all coefficients will be transferred; *xmem_rd_addr* and *xmem_wr_addr* are pointers to the external memory that

Algorithm 1 Pseudocode for Beamforming

1: {*configure the number of input channels available*}
2: **ConfC** (C);
3: {*disable all BeamFormers until system is configured*}
4: **InStreamEn** (DISABLE_INPUTS_MASK);
5: {*clear the contents of all SPRs*}
6: **ClrSPRs** ();
7: {*configure decimators size*}
8: **DFirF** (DECIMATOR_SIZE, DECIMATOR_TYPE);
9: {*configure H(z) filters size*}
10: **DFirF** (H_SIZE, H_TYPE);
11: {*configure interpolator size*}
12: **DFirF** (INTERPOLATOR_SIZE, INTERPOLATOR_TYPE);
13: {*configure the samples addresses*}
14: **SSA** (SamplesAddr);
15: {*transfer all H(z) coefficients to on-chip buffers*}
16: **BufCoef** (CoefXMemAddr, BufAddr);
17: {*load the coefficients to all decimators and interpolator*}
18: **LdCoef** (BufAddr);
19: {*initialize external memory reading and writing pointers*}
20: xmem_rd_addr=INPUT_DATA_XMEM_ADDR;
21: xmem_wr_addr=OUTPUT_DATA_XMEM_ADDR;
22: {*enable BeamFormers*}
23: **InStreamEn** (ENABLE_INPUTS_MASK);
24: {*process streaming data*}
25: **while** (1) **do**
26: **BFSrc** (aper, xmem_rd_addr, xmem_wr_addr);
27: {*update external memory pointers*}
28: xmem_rd_addr=xmem_rd_addr+1024·C;
29: xmem_wr_addr=xmem_wr_addr+1024;
30: **end while**

read input channels data and write source samples respectively. *INPUT_DATA_XMEM_ADDR* is an external memory address, where input channels data are stored, while *OUTPUT_DATA_XMEM_ADDR* is an external memory address, where samples of extracted sources are written back. Finally, *aper* is the current source aperture.

The pseudocode starts by configuring the number *C* of available input channels, and then disabling them from processing, since the system is not yet properly setup. All SPRs are initialized and then the decimator, H(z) and interpolator filter sizes are configured. Afterwards, the addresses of all samples within the on-chip buffers are specified, and all required decimator, H(z) filters and interpolators coefficients are distributed from the external memory to on-chip buffers. Once this step is completed, the decimators and interpolator coefficients are reloaded. Since the system is now configured, all BeamFormers are enabled. Finally, in each iteration of the while-loop, the current source aperture *aper* is used and 1024·C samples are read to extract 1024 source samples, which are written to the external memory. Both external memory read/write pointers are updated accordingly for the next while-loop iteration.

TABLE III
COMPARISON BETWEEN THE SOFTWARE AND HARDWARE IMPLEMENTATIONS.

# of sources	t_{SW-OMP}(msec)	t_{MC-BFP}(msec)	Speedup
4	10640	3127	3.40
8	17497	6251	2.80
12	24558	9375	2.62

In order to evaluate the proposed architecture, we implemented it in our hardware prototype, presented in [7], which was significantly improved with high-level programming abilities. Our implementation was mapped onto a Virtex4 FX60 FPGA device. As host processor we used one of the two integrated PowerPC cores. We compared our prototype against an OpenMP-annotated software implementation on a Core2 Duo processor at 3.0 GHz. Table III provides the Core2 execution time (t_{SW-OMP}), the MC-BFP execution time (t_{MC-BFP}) and the obtained speedup under different number of sources to be extracted. As we can observe, our proposed approach can extract 12 sources 2.6 times faster compared to the Core2 Duo approach.

V. Conclusions and Future Work

In this paper, we proposed a minimal architecture for audio-Beamforming applications. Our approach combines software programming flexibility with improved hardware performance. We demonstrated how the proposed instructions can be used to develop a compact, yet efficient program, which can be applied to control a reconfigurable multi-core processor for Beamforming applications. Finally, we evaluated our proposal using an FPGA prototype, which was found able to extract in real-time up to 12 acoustic sources 2.6x faster than a Core2 Duo solution.

Acknowledgments

This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), FP7 Reflect (grant 248976).

References

[1] Gerrit Blaauw and Frederick Brooks, "Computer Architecture: Concepts and Evolution," February 1997.
[2] B. V. Veen and K. Buckley, "Beamforming: a versatile approach to spatial filtering," in *IEEE ASSP Magazine*, vol. 5, April 1988, pp. 4–24.
[3] Bill Kapralos et. al., "Audio-visual localization of multiple speakers in a video teleconferencing setting," in *International Journal of Imaging Systems and Technology*, vol. 13(1), October 2003, pp. 95–105.
[4] Ka-Fai Cedric Yiu et. al., "Reconfigurable acceleration of microphone array algorithms for speech enhancement," in *Application-specific Systems, Architectures and Processors*, 2008, pp. 203–208.
[5] Zohra Yermeche et. al., "Real-time implementation of a subband beamforming algorithm for dual microphone speech enhancement," in *IEEE International Symposium on Circuits and Systems*, May 2007, pp. 353–356.
[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] D. Theodoropoulos et. al., "A Reconfigurable Beamformer for Audio Applications," in *IEEE Symposium on Application Specific Processors*, pp. 80–87.