Custom Architecture for Multicore Audio Beamforming Systems

DIMITRIS THEODOROPOULOS and GEORGI KUZMANOV, Delft University of Technology
GEORGI GAYDADJIEV, Delft University of Technology and Chalmers University of Technology

The audio Beamforming (BF) technique utilizes microphone arrays to extract acoustic sources recorded in a noisy environment. In this article, we propose a new approach for rapid development of multicore BF systems. Research on literature reveals that the majority of such experimental and commercial audio systems are based on desktop PCs, due to their high-level programming support and potential of rapid system development. However, these approaches introduce performance bottlenecks, excessive power consumption, and increased overall cost. Systems based on DSPs require very low power, but their performance is still limited. Custom hardware solutions alleviate the aforementioned drawbacks, however, designers primarily focus on performance optimization without providing a high-level interface for system control and test. In order to address the aforementioned problems, we propose a custom platform-independent architecture for reconfigurable audio BF systems. To evaluate our proposal, we implement our architecture as a heterogeneous multicore reconfigurable processor and map it onto FPGAs. Our approach combines the software flexibility of General-Purpose Processors (GPPs) with the computational power of multicore platforms. In order to evaluate our system we compare it against a BF software application implemented to a low-power Atom 330, a middle-ranged Core2 Duo, and a high-end Core i3. Experimental results suggest that our proposed solution can extract up to 16 audio sources in real time under a 16-microphone setup. In contrast, under the same setup, the Atom 330 cannot extract any audio sources in real time, while the Core2 Duo and the Core i3 can process in real time only up to 4 and 6 sources respectively. Furthermore, a Virtex4-based BF system consumes more than an order less energy compared to the aforementioned GPP-based approaches.

Categories and Subject Descriptors: B.7.1 [**Hardware**]: Algorithms Implemented in Hardware

General Terms: Design, Performance

Additional Key Words and Phrases: Immersive audio, reconfigurable computing, audio beamforming, reconfigurable processors, embedded processors

ACM Reference Format:
Theodoropoulos, D., Kuzmanov, G., and Gaydadjiev, G. 2013. Custom architecture for multicore audio beamforming systems. ACM Trans. Embedd. Comput. Syst. 13, 2, Article 19 (September 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2514641.2514646

1. INTRODUCTION

Record of an accurate aural environment has been studied for many decades, which led to the development of various different techniques for sound acquisition. Efficient microphones placement has been well studied, because it directly affects the

D. Theodoropoulos is now with the Institute of Computer Science, Foundation for Research and Technology, FORTH, Heraklion, Greece.

This research is partially supported by Artemisia project (grant no. 100203), Artemisia SMECY (grant no. 100230), FP7 Reflect (grant no. 248976), FP7 DeSyRe (grant no. 287611), and FP6 hArtes (IST-035143).

Authors' addresses: D. Theodoropoulos (corresponding author) Institute of Computer Science, Foundation for Research and Technology – Hellas (FORTH), Heraklion, Crete, Greece; email: dtheodor@ics.forth.gr; G. Kuzmanov and G. Gaydadjiev, Computer Engineering Laboratory, EEMCS, Delft University of Technology, P.O. Box 5031, 2600 GA Delft. The Netherlands.

DOI: http://dx.doi.org/10.1145/2514641.2514646

Signal-to-Noise Ratio (SNR). In general, sound recording techniques can be divided into four main approaches.

(1) *Acquire the speech signal directly from the source.* This approach is suitable for applications where carrying a close-talk recording device is acceptable, like music concerts and live TV broadcastings.
(2) *Surround recording.* This technique is followed when carrying recording devices is not an acceptable solution. An exemplary case is movie actors, where microphones should not be visible.
(3) *Recording of the signals that reach the ears (binaural signals).* This method implies putting two microphones facing away from each other at a distance equal to the one between human ears (approximately 18 cm). It is applicable in cases where the recorded signals will be rendered through headphones.
(4) *Utilize microphone arrays to amplify the original acoustic source.* This solution is applicable in cases where distant speech signals need to be extracted and attenuate any ambient noise. Example applications are advanced teleconference products and surveyance systems inside public areas (like airports or public stations), where the security personnel can record and acquire the speech signals of suspects.

The first three techniques have been used for many decades, because they require the least complex hardware setup. However, they introduce particular shortcomings. In the first technique, for example, although it is well established for performers and presenters to carry a wired recording device, it still requires complex cable setups within the performance area. Even in the case of a wireless microphone, it is considered uncomfortable to constantly carry it. The second approach employs a small number of microphones to record "sound images" [Snow 1955] of the area and not directly speech signals. Thus, there can be cases where the SNR is low, leading to poor audio quality. The binaural recording method [Kyriakakis 1998] offers high sound localization and perception quality, however, it requires that the listener wears headphones. Although there are systems, called Ambiophonics [Farina et al. 2001], that address this shortcoming, still there are movement restrictions imposed within a small listening area [Mouchtaris et al. 2000].

The last technique is called BeamForming (BF) [Veen and Buckley 1988] and has already been widely used for many decades in different application fields, like the SOund Navigation And Ranging (SONAR), RAdio Detection And Ranging (RADAR), telecommunications, and ultra-sound imaging [Wall and Lockwood 2005]. Over the last years, the BF technique has also been adopted by the audio research society, mostly to enhance speech recognition. The main advantage is that any stationary or moving audio source within a certain noisy area can be efficiently isolated and extracted with high SNR. Furthermore, there is no need for carrying any recording device. The BF technique requires the utilization of microphone arrays which capture all emanating sounds. All incoming signals are then combined to amplify the primary source signal, while at the same time suppressing any environmental noise. However, due to the increased number of input channels compared to other approaches, its main shortcoming is that it requires substantial signal computations, thus powerful processing platforms.

Figure 1 shows the different number of microphones that each recording technique requires. As depicted, in general, surround recording techniques employ no more than five microphones, one of each recorded channel [Theile 2001]. Binaural recordings use only two microphones, one for each ear, while in the case of a close-talk recording, each speaker uses a single device. In contrast, nowadays there are commercial and experimental systems that utilize the BF technique and employ from tens to more than 1000 microphones [Weinstein et al. 2004].

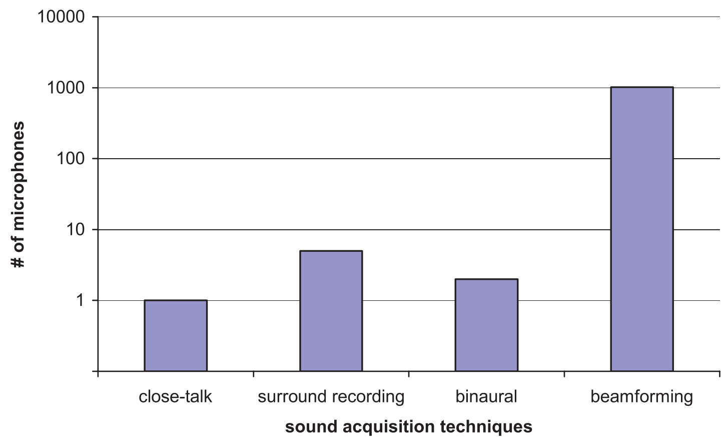


Fig. 1. Number of utilized microphones among different sound acquisition techniques.

As discussed before, the BF technology alleviates the majority of the shortcomings that other recording techniques introduce, at the expense of an increased number of input channels. Moreover, it is highly scalable, thus can be applied to future consumer and professional multimedia and telecommunication products, ranging from portable devices to high-quality teleconference systems. Consequently, because of its inherent parallelism, the most suitable implementation hardware platform domain is the one of multicore devices that integrate a large number of processing cores.

However, research on literature reveals that the majority of experimental and commercial BF systems are based on standard PCs, due to their high-level programming support and potential of rapid system development. It is well accepted that today's software languages provide a very intuitive development environment that allows rapid systems prototyping and implementation. However, these approaches introduce the following drawbacks.

—*Performance bottlenecks*. General-Purpose Processors (GPPs) provide limited computational power, thus in many cases additional PCs are utilized to efficiently drive all input channels.
—*Excessive power consumption*. Contemporary high-end GPPs consume tens of Watts of power when they are fully utilized. Furthermore, when additional PCs are employed to drive all required channels, the total system power consumption easily exceeds the kWatt scale.
—*Increased overall system cost*. Utilization of many PCs leads to an approximately linear overall system cost increase, thus constraining the employment of such systems only to professional applications or large academic projects.

To partially address the aforementioned problems, researchers have considered alternative hardware platforms to implement immersive audio systems. Various systems have been developed based on Digital Signal Processors (DSPs), in order to reduce power consumption, however, performance is still limited. In contrast, recent Graphic Processor Unit (GPU) -based BF approaches provide a significantly better performance compared to PC-based systems, however, a considerable effort is required, in order to efficiently analyze and map the application onto all processing resources. Custom hardware solutions alleviate both of the aforementioned drawbacks. However, in the majority of cases, designers are primarily focused on just performing all required calculations faster than a GPP. Such approaches do not provide a high-level interface for testing the system that is under development.

The main contribution of this article is the analytical presentation and evaluation of a custom BF architecture, originally proposed in Theodoropoulos et al. [2010]. The microarchitectural support is based on our previously presented reconfigurable BF processor [Theodoropoulos et al. 2009], and is specifically tailored to reconfigurable multicore implementations. We prove that our proposal combines the programming flexibility of software approaches with the high performance, low power consumption, and limited memory requirements of reconfigurable hardware solutions. The architecture implementation allows utilization of various number of processing elements, therefore it is suitable for mapping onto reconfigurable technology. With respect to the available reconfigurable resources, different FPGA implementations with different performances are possible, where all of them use the same architecture and programming paradigm. More specifically, the contributions of this article are the following.

—*A high-level architecture for BF audio applications.* We propose a high-level architecture that consists of 9 instructions, which allow customization and control of BF audio systems implemented to FPGAs. Our proposal employs a logically shared, physically distributed memory hierarchy and allows a high-level interaction with different processing elements.

—*Microarchitectural support for reconfigurable processors.* We describe our microarchitectural support for the proposed architecture, which allows the utilization of customizable number of processing elements, thus making it suitable for systems based on reconfigurable technology.

—*Our reconfigurable design demonstrates high performance under different input scenarios.* We conducted various tests against an OpenMP-annotated software approach with SSE2 extensions enabled that was run on a low-power Atom 330 at 1.6 GHz, a middle-ranged Core2 Duo at 2.8 GHz, and a high-end Core i3 at 3.1 GHz. Experimental results suggest that a Virtex4-based and a Virtex6-based design with 16 input channels can extract in real time up to 14 and 16 acoustic sources respectively. In contrast, under the same number of microphones, the Atom-330-based system could not extract in real time acoustic sources, while the Core2 Duo and the Core i3 GPPs were limited up to 4 and 6 sources respectively.

—*Our implementations achieve low energy consumption.* Based on the processing time and the power dissipation, we estimate the energy consumption of each considered platform. Results suggest that a Virtex4-based implementation requires more than an order of magnitude less energy compared to the GPP-based systems.

The rest of the article is organized as follows: Section 2 provides a short theoretical background on the BF technique and presents various systems that utilize it. In Section 3 we propose our custom BF architecture with its dedicated memory organization and supported instruction set. Section 4 describes the microarchitectural support of our custom architecture, while in Section 5 we provide experimental results under different input channels and sources scenarios. Finally, Section 6 concludes.

2. BACKGROUND AND RELATED WORK

Theoretical background. Generally, there are two different types of BF: nonadaptive (or time invariant or nonblind) and adaptive (or blind) [Sallberg et al. 2006; Veen and Buckley 1988]. Nonadaptive methods are based on the fact that the spatial environment is already known and tracking devices are used to enhance speech recognition. In contrast, adaptive approaches do not utilize tracking devices to locate the sound source. In fact, the received signals from the microphones are used to calibrate properly the beamformer, in order to improve the quality of the extracted source. In the audio domain, in the majority of the cases a nonadaptive delay-and-sum approach is utilized

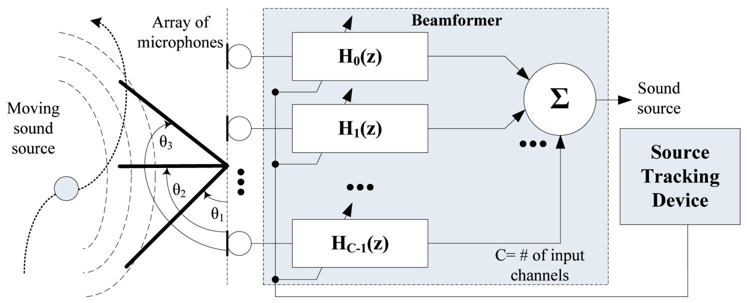


Fig. 2. A filter-and-sum beamformer.

[Veen and Buckley 1988], due to its rather simple implementation and because a tracking device (such as a video camera) is almost always available.

The term of beamformer refers to a processor that performs spatial filtering in order to estimate a signal arriving from a particular location. Thus, even in the case where two signals contain overlapping frequencies, a beamformer is able to distinguish each one of them, as long as they originate from different locations. Figure 2 depicts a schematic overview of a beamformer utilizing the filter-and-sum approach [Veen and Buckley 1988]. As we can see, the system consists of an array of microphones sampling the propagating wavefronts. Each microphone is connected to an FIR filter $H_i(z)$, while all filtered signals are summed up to extract the desired audio source. Although not always required, in many cases the input data channels are downsampled by a factor D in order to reduce the data rate.

$$xD_i[n] = x_i[n \cdot D] \tag{1}$$

Here x_i is the input signal, xD_i is the downsampled signal, $i = 0 \ldots C - 1$ and C is the number of input channels (microphones). Each downsampled signal is filtered using a particular coefficients set based on the source location

$$yD_i[n] = \sum_{j=0}^{H-1} h_i[j] \cdot xD_i[n-j], \tag{2}$$

where H is the number of filter taps and h are the filter coefficients. The beamformer output is given by the sum of all yD_i signals

$$yD[n] = \sum_{i=0}^{C-1} yD_i[n], \tag{3}$$

where yD is the downsampled extracted source. Then, yD is upsampled by the factor L (in most cases L = D) according to Eq. (4) to acquire the upsampled extracted source y.

$$y[n] = \begin{cases} yD\left[\frac{n}{L}\right], & if\ \frac{n}{L} \in Z \\ 0, & otherwise \end{cases} \tag{4}$$

The idea is to use the FIR filters as delay lines that compensate for the introduced delay of the wavefront arrival at all microphones [Kapralos et al. 2003]. The combination of all filtered signals will amplify the desired one, while all interfering signals will be attenuated. However, in order to extract a moving acoustic source, it is mandatory to reconfigure all filters coefficients according to the source current location. For example, as illustrated in Figure 2, 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 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. Over the last years, various systems that utilize GPUs under different application domains have been published in the literature. In Wall and Lockwood [2005] the authors describe a hybrid approach that utilizes 14 Virtex4 LX25 FPGAs and a GPU connected to a desktop PC to perform 3D-parallel BF and scan conversion for real-time ultrasonic imaging. Input data are received from 288 channels that are connected to analog-to-digital converters. Digitized data are forwarded to the FPGAs, which calculate the signal delay, interpolation, and apodization. All processed data are transferred though the PCI from the FPGAs to the GPU. In Nilsen and Hafizovic [2009], the authors utilize a GeForce 8800 GPU to design a delay-and-sum beamformer in the time and frequency domain. To evaluate their designs they perform experiments under different number of input channel setups ranging from 79 to 1216. According to the results, a time domain and a frequency domain beamformer can achieve speedup up to 12x and 15x respectively compared to a Xeon quad-core processor.

In the *audio* domain, the BF technique is widely used in hand-held devices, like cell phones and personal digital assistants. Such embedded systems introduce many constraints regarding computational resources and power consumption. To alleviate these problems, the authors in Mihov et al. [2008] designed a time-invariant beamformer tailored to small devices that consists of two microphones. According to the paper, results suggest an SNR improvement of 14.95 dB when using two microphones, instead of one. A data-driven beamformer for a binaural headset is presented in Tashev and Seltzer [2008]. The authors integrate two microphones to the headphones and employ a head and torso simulator to acquire the source signal for BF. The improvement of SNR is in the range between 4.4 and 6.88 dBC.

Commercial products for audio BF have been developed by various companies. For example, Squarehead [Squarehead Technology 2013] develops the audioscope, a dual-core PC-based system, that employs 300 omnidirectional microphones for audio capturing. Another company, called Acoustic Camera, develops PC-based BF system that utilize sound acquisition arrays ranging from few tens to more than a hundred elements. Polycom and Microsoft presented the *CX5000* unified conference station [Polycom Inc. 2009], which is the latest version of the Roundcam, originally presented in Cutler et al. [2002]. Roundcam consists of five built-in cameras that offer a 360^o panoramic view of the conference room and eight microphones to capture the speech signals. It connects to a dual CPU 2.2 GHz Pentium 4 workstation through a Firewire bus. All image and sound processing is done to the workstation. For computational efficiency and low latency, the authors utilize a delay-and-sum beamform approach. Lifesize is another company that produces high-quality communication systems. For example, the *LifeSize Focus* teleconferencing camera supports high-definition video and uses two omnidirectional microphones to capture audio sources using BF. A small set of these cameras is utilized in the company's advanced communication systems, like the *LifeSize Room* series, to record image and transmit it to the remote location. Sound sources are rendered to the remote location using high-definition audio.

In Cedrick [2005], the author presents the NIST Mark-III microphone array that can be used for speech enhancement and recognition. The proposed platform utilizes 64 input channels that are connected to a Spartan II FPGA via analog-to-digital converters. The FPGA is connected through Ethernet to a host desktop PC that runs the NIST Smart Flow II software platform [Mei et al. 2006; Fillinger et al. 2007]. The latter employs a Web camera that identifies a speaker's face and steers accordingly the BF, in order to enhance the speech signal and attenuate any ambient noise.

The authors of Yiu et al. [2008] present a hardware accelerator that utilizes microphone array algorithms based on the use of calibrated signals together with subband processing. The proposed design utilizes a frequency domain modified recursive least-squares adaptive algorithm and the SNR maximization of the BF algorithm. Up to 7 instances of the proposed design can fit in a Virtex4 SX55 FPGA, achieving a speedup of up to 41.7x compared to the software implementation.

A similar approach is chosen in Xcell [2007] where a real-time beamformer mapped on an FPGA platform is presented. The BF engine is based on the QR matrix decomposition (QRD). In each update of the beamformer, new input samples are generated by a Matlab host application and forwarded to the FPGA, where the QRD engine processes them. Once processing is done, the new weight vector is returned back to the host processor and a new chunk of data is forwarded to the FPGA. The complete design occupies 3530 Virtex4 slices and requires 56.76 μsec to decompose a 10×10 matrix at 250 MHz.

A Digital Signal Processor (DSP) implementation of an adaptive subband BF algorithm is presented in Yermeche et al. [2007], known as the Calibrated Weighted Recursive Least-Squares (CWRLS) beamformer. The authors utilize an analog devices ADSP21262 DSP processor [Analog Devices 2004] to perform CWRLS-based BF over a two-microphone array setup. According to the paper, results indicate that there is an up to 14 dB SNR improvement, but the computational load of the DSP processor can be up to 50% with two input channels. The presented implementation is also energy efficient, since it was predicted to have an operation time of up to 20 hours, under the aforementioned processor utilization.

An experimental video teleconferencing system is presented in Fiala et al. [2004]. The authors combine an omnidirectional video camera and an audio BF microphone array into a device that is placed in the center of a meeting table. Nonstationary participants are identified with computer vision algorithms and their speech is recorded from a circular 16-microphone array. Audio processing is done using a TMS320C6201 DSP processor [Texas Instruments 2002] at 11.025 kHz sampling rate.

In Beracoechea et al. [2006], the authors describe an immersive audio system that consists of 12 linearly placed microphones. The sound source is tracked through audio and video tracking algorithms, while the beamformer is steered accordingly. The audio signal is extracted through BF and encoded using the MPEG2-AAC or G722 encoders. The encoded signal is received from a second remote PC and the audio signal is rendered using the Wave Field Synthesis (WFS) technology [Berkhout et al. 1993] through a 10-loudspeaker array.

A similar system is presented in Teutsch et al. [2003]. The authors describe a real-time immersive audio system that exploits the BF technique and the WFS technology. The system performs sound recording from a remote location A, transmits it to another one B, and renders it through a loudspeaker array utilizing WFS. The complete system consists of 4 PCs, out of which one is used for the WFS rendering, one for BF, one for the source tracking, and one as a beamsteering server.

In addition, the work presented in Buchner et al. [2002] addresses the problem of echo cancellation that is inherent to contemporary multimedia communication systems. The authors propose a strategy to reduce the impact of echo while transmitting the recorded signal to a remote location. The idea is to apply the proposed Acoustic Echo Cancellation (AEC) to the "dry" source signals that will be rendered through the loudspeaker array. Then, the AEC output signals are subtracted from the output signals of the beamformer's time-invariant components. In order to test their approach, the authors develop a real-time implementation using a standard desktop PC that consists of 11 microphones and 24 loudspeakers.

Finally, nowadays there are many projects that utilize different microphone array sizes and setups. One of the most famous implementations is the Large AcOUstic Data (LOUD) [Weinstein et al. 2004], which was part of the MIT Oxygen project [MIT CSAIL: MIT Project Oxygen 2004]. The LOUD microphone array consists of 1020 elements arranged into a 2D planar setup and produces data at a rate of 50 MB/sec. All data are streamed to a custom-designed tiled parallel processor, based on the raw ISA [Taylor et al. 2001, 2002, 2004]. Experimental results suggest that utilizing such a large microphone array can dramatically improve the source recognition accuracy up to 90.6%.

3. CUSTOM ARCHITECTURE

BF instruction set. The design of BF systems requires various tests before their final implementation. Based on the size of the recording area and the hardware cost limitations, the designer has to evaluate the SNR quality of the extracted sources under different numbers of microphones. Furthermore, internal signal calculations, except filtering, in many cases require also 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. For example, one coefficient set can provide a better filtered signal than another set, under the same number of filter taps. It is important for the designer to have the option to rapidly change and evaluate each filter coefficients set. In addition, many tests should be conducted to decide the number of source apertures into which the recording area should be divided. Such tests, when developing a software beamformer, are easily applicable, however, it is not the case when custom hardware solutions are required. In the latter case, the designers should also be able to easily perform tests under different source apertures.

The main goal of the proposed architecture is to provide a certain instruction set that will allow easy customization of many vital system parameters, efficient audio data processing, and system debugging through a high-level interface for reconfigurable BF audio systems. Furthermore, these instructions should be platform independent and hide any implementation details, thus allowing the same program to be executed to different FPGA implementations. After studying the BF technique, we concluded with the following requirements regarding the kind of instructions that the programmer should have at his/her disposal.

(1) *Enable or disable input channels.* In order to allow easy and fast input channel tests, we decided to provide an instruction that would allow the programmer to disable/enable them in any arbitrary way. Consequently, it would assist on rapidly fine-tuning and deciding the number of required input channels for the entire system.
(2) *System configuration.* It is very important to provide instructions to the user that would allow a high-level configuration of many key system parameters. This way, all implementation details for system customization can be hidden, thus assisting on easy and rapid development. Examples of these parameters can be the size and coefficient sets specification for various digital filters, and the number of input channels.
(3) *Efficient audio data processing.* A specific instruction is required to control data processing of audio samples. Although all computations will be performed in parallel from many processing elements, the user should be completely isolated from any platform-specific details. Moreover, a simple high-level interface should provide all required parameters, which internally will initiate massively parallel audio data processing.

Table I. Supported Instructions by the Proposed Architecture

Instruction type	Full name	Mnemonic	Parameters
I/O	Input Stream Enable	InStreamEn	*b_mask*
System setup	Clear SPRs	ClrSPRs	NONE
	Declare FIR Filter size	DFirF	FSize, FType
	Specify Samples Addresses	SSA	buf_sam_addr
	Buffer Coefficients	BufCoef	xmem_coef_addr, buf_coef_addr
	Load Coefficients	LdCoef	buf_coef_addr
	Configure C	ConfC	C
Data processing	BF Source	BFSrc	*aper*, xmem_read_addr, xmem_write_addr
Debug	Read SPR	RdSPR	SPR_num

(4) *Debugging capabilities*. Efficient system debugging considerably assists on rapid development. For this reason, an instruction that would allow the user to check important system parameters should be supported by our architecture.

Taking into account the aforementioned requirements, the proposed architecture consists of eight high-level instructions that can be used to configure an immersive audio system, and start, manage, or stop processing of input and output data. Furthermore, there is an additional 9th instruction that can be used for debugging purposes. In order to support many system setups, we provide a versatile environment that allows the adjustment of various parameters. For example, an audio acquisition module that utilizes the BF technique may have any number of source apertures that can be identified. Furthermore, the BF FIR filters can consist of any number of taps. For these reasons, the proposed programming architecture was designed in such a way that the programmer can conveniently change these vital system parameters.

Table I shows the nine instructions, divided into four categories, namely *I/O*, *system setup*, *data processing*, and *debug*. The *I/O* instruction is used to enable or disable audio streaming to input 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 user in order to read a certain set of Special-Purpose Registers (SPRs) that are used to store many system parameters. In the following, we describe each of the instructions. We assume that an immersive audio system consists of C input channels.

InStreamEn. This enables or disables streaming of audio samples from input channels to the BF processing units.

ClrSPRs. This clears the contents of all SPRs.

DFirF. This declares the size of a filter to the system.

SSA. This specifies the addresses from which all input samples are read.

BufCoef. This fetches all decimator and interpolator coefficients from external memory to on-chip buffers.

LdCoef. This distributes all decimator and interpolator coefficients to the corresponding filters in the system.

ConfC. This defines the number of input channels that are available to the system.

BFSrc. This 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.

RdSPR. This used for debugging purposes and allows the programmer to read any of the SPRs.

Memory and registers organization. The application of our custom architecture to reconfigurable devices comprises dedicated register organization and distributed memory

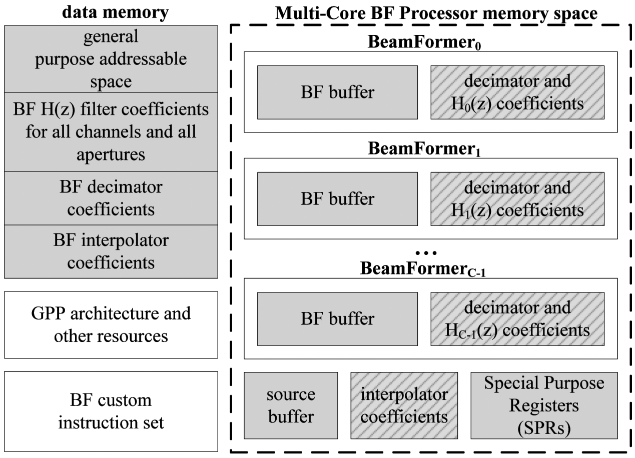

data memory
general purpose addressable space
BF H(z) filter coefficients for all channels and all apertures
BF decimator coefficients
BF interpolator coefficients
GPP architecture and other resources
BF custom instruction set
Multi-Core BF Processor memory space
BeamFormer0
BF buffer
decimator and H0(z) coefficients
BeamFormer1
BF buffer
decimator and H1(z) coefficients
...
BeamFormerC-1
BF buffer
decimator and HC-1(z) coefficients
source buffer
interpolator coefficients
Special Purpose Registers (SPRs)

Fig. 3. Memory organization for BF applications.

Table II. Special-Purpose Registers Mapping for BF

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, $i = 0 \ldots C-1$
SPR[10+C] − SPR[$9+2\cdot C$]	channel i 1024 samples buffer address, $i = 0 \ldots C-1$

buffers. An important feature of the architecture is that it is based on a multicore processing paradigm. This allows the design of scalable microarchitectures, with respect to the available hardware resources, which makes the architecture suitable for reconfigurable implementations.

Figure 3 illustrates the logical organization of the memory and the registers of the proposed architecture when utilizing the BF technique. It is assumed that it operates as an architectural extension of a GPP in a coprocessor paradigm. The architecture assumes multicore 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 runtime. The latter option makes it suitable for implementations on platforms with partial configuration capabilities. The host GPP and our custom *MultiCore BeamForming Processor* (MC-BFP) exchange synchronization parameters and memory addresses via a set of SPRs, shown in Table II. Each *beamformer* module has an on-chip *BF 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 memory organization that is considered by the proposed architecture is the user-accessible memory space, as illustrated in Figure 3. The nonuser addressable space is annotated with the stripe pattern. In order to provide a high-level programming environment, the programmer has read and write access to the *BF buffers*, the *source buffer*, the external memory, and the GPP 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 *BF buffers* to the decimators, H(z) filters, and interpolator. This way the user avoids completely any low-level interaction with the hardware platform.

Instructions parameters analysis. InStreamEn: 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 channel streaming by setting 0 or 1 to its value, respectively. The binary mask is stored in SPR0, as shown in Table II.

ClrSPRs. It does not require any parameters.

DFirF. It writes the size of a filter 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. Its parameter is an array of pointers *buf_sam_addr* to the starting address of all on-chip *BF buffers*. *SSA* writes from SPR[10+C] to SPR[9+2·C] the on-chip *BF buffers* starting addresses. Furthermore, it writes to SPR7 the *source buffer* address, where 1024 samples of the extracted source signal are stored, as shown in Table II.

BufCoef. Its parameters are 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 *BF buffers*, where all coefficients will be stored. *BufCoef* does not write any values to SPRs.

LdCoef. 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. Once all filter coefficients are transferred, *LdCoef* writes a done flag to SPR4, as shown in Table II.

ConfC. 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, as shown in Table II.

BFSrc. It 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 $i = 0, \ldots, C-1$, thus channel processing is performed concurrently. Once all data calculations are finished, a done flag is written to SPR6, as shown in Table II.

RdSPR. It requires as parameter the number of SPR *SPR_num* that needs to be read.

3.1. Programming Model

In Algorithm 1, we illustrate through pseudocode how to set up a BF 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 previously. 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

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
```

to specify the filter type. *SamplesAddr* is an array of pointers to each on-chip *BF 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 *BF buffers*, where all coefficients will be transferred. *xmem_rd_addr* and *xmem_wr_addr* are pointers to the external memory that 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 in line 2 by using the *ConfC* instruction to configure the number *C* of currently available input channels to the BF system. In line 4, all input channels are disabled from processing using the *InStreamEn*, since the system is not yet properly set up. In line 6 all SPRs are initialized by clearing their contents. In lines 8, 10, and 12, the *DFirF* instruction is used to configure the decimator, H(z), and interpolator filter sizes. In line 14, the addresses of all samples within the on-chip *BF buffers* are specified, using the *SSA* instruction. In line 16, the *BufCoef* distributes all required decimators and interpolator coefficients from the external memory to on-chip *BF buffers*. Once it is done, in line 18, the *LdCoef* instruction is used to configure the decimators and interpolator coefficients. In line 20, *xmem_rd_addr* is initialized

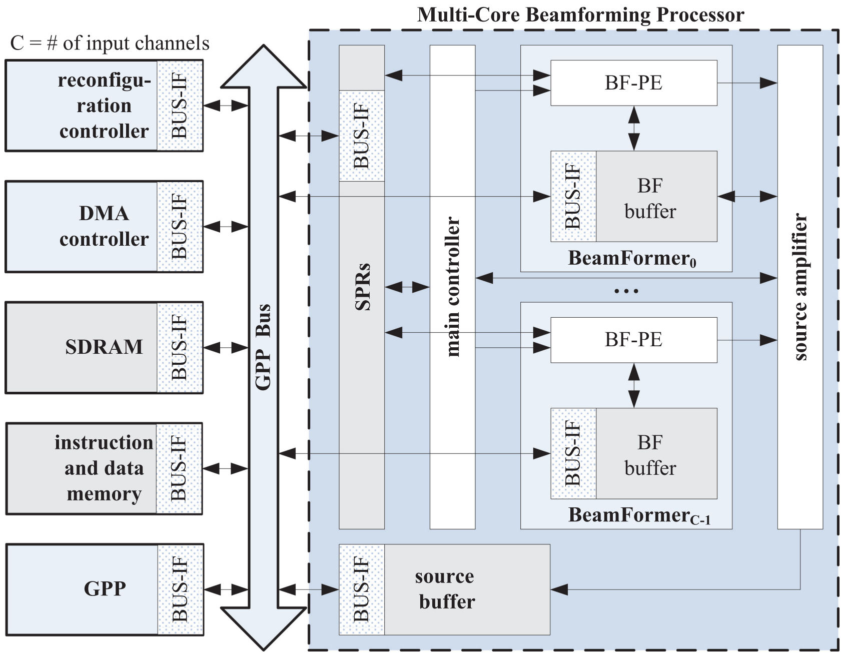


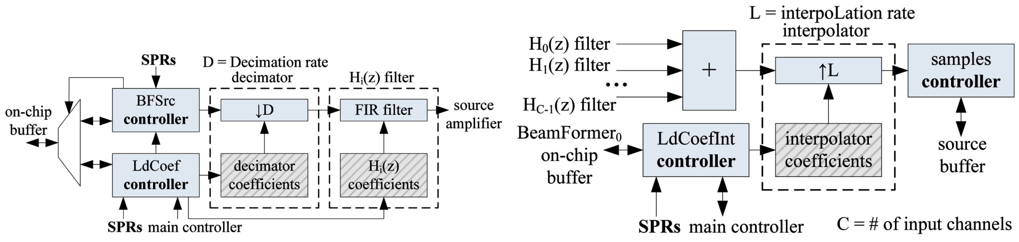
Fig. 4. Multicore implementation of the BF system.

pointing at the input data stored in the external memory. In line 21, *xmem_wr_addr* points to the external memory address where the extracted source samples are stored. Once the system is properly configured, all beamformers are enabled in line 23. Finally, in each iteration of the while-loop in line 25, the current source aperture *aper* is used and 1024·C samples are read to extract 1024 source samples using the *BFSrc*, which are written to the external memory. The *xmem_rd_addr* and the *xmem_wr_addr* pointers are increased by 1024·C and 1024 entries, respectively, to properly point to the required input and output external memory locations for the next iteration.

It should be noted that any time, the designer can use the *RdSPR* instruction for debugging purposes. Also, if the user wants to perform additional experiments under different number of microphones, it can be done by reconfiguring the system using the *ConfC* instruction. Furthermore, in case the designer wishes during runtime to test different coefficient sets or increase/decrease the total number of source apertures, it is possible by just providing a new array of pointers to the *BufCoef* instruction. The *LdCoef* can then be used to reload all decimators and interpolator with the new coefficients sets, while the *BFSrc* instruction will extract sources based on the new H(z) coefficients sets.

4. RECONFIGURABLE ARCHITECTURE IMPLEMENTATION

MultiCore BF Microarchitecture. Our system is based on our reconfigurable BF design, originally presented in Theodoropoulos et al. [2009], however, considerable improvements and design enhancements have been applied, in order to support the proposed custom architecture. Figure 4 illustrates the multicore implementation of our BF architecture. As mentioned in Section 3, the latter is assumed that it operates as an architectural extension of a GPP in a coprocessor paradigm. The architecture assumes multicore processing, distributed among *C* processing modules that process data from *C* input channels. A *GPP bus* is used to connect the on-chip GPP memory and external



(a) the BF processing element (BF-PE) structure (b) the source amplifier structure

Fig. 5. The BF-PE and source amplifier structures.

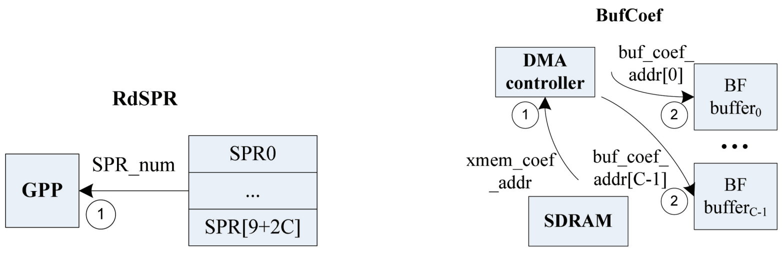
SDRAM with the GPP via a standard bus interface (BUS-IF). Furthermore, in order to accelerate data transfer from the SDRAM to on-chip *BF buffers*, we employ a *Direct Memory Access (DMA) controller*, which is also connected to the same bus. A partial reconfiguration controller is employed to provide the option of reloading the correct bitstreams based on the currently available number of input channels. All user addressable spaces inside the *MC-BFP*, like *SPRs*, *BF 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 all decimators and the interpolator. Furthermore, it enables input data processing from all channels, and acknowledges the GPP as soon as all calculations are done.

Each *beamformer* module consists of one *BF buffer* and a *BeamForming Processing Element* (BF-PE), which is illustrated in Figure 5(a). As can be seen, there is a *LdCoef controller* and a *BFSrc controller*. Based on the current source aperture, the former is responsible for reloading the required coefficients sets from the *BF buffer* to the decimator and H(z) filter. The *BFSrc controller* reads 1024 input samples from the *BF buffer* and forwards them to the decimator and the H(z) filter.

All *beamformer* modules forward the filtered signals to the *source amplifier*, which is shown in Figure 5(b). The *LdCoefInt controller* is responsible for reloading the coefficients set to the interpolator. As we can see, all $H_i(z)$ signals, where $i = 0, \ldots, C - 1$, are accumulated to strengthen the original acoustic source signal, which is then interpolated. Finally, the *samples controller* is responsible for writing back to the *source buffer* the interpolated source signal.

BF Data Processing Flow. Figure 7 illustrates how BF data processing is divided among C *beamformers*, under a C-sized microphone array setup. In each iteration, $1024 \cdot C$ samples are fetched from the SDRAM and stored to the on-chip *BF buffer* of each *beamformer*. once data transfer is done, all *beamformers* start processing concurrently the audio samples. More specifically, each one of them downsamples the recorded signals by a factor D. The downsampled signals are forwarded to the H(z) BF filters, and all outputs are accumulated in order to strengthen the original acoustic source. The latter is upsampled by a factor L and the result is stored to the external memory. The process is repeated for each i acoustic source within the recording area, where $i = 0, \ldots, S - 1$ and S is the total number of sources. As soon as a 1024-sample chunk is extracted for all S sources, the recorded data of a new 1024-sample time frame is loaded from the SDRAM to the *BF buffers* for further processing.

BF Instruction Implementation. All *SPRs* are accessible from the GPP, because they belong to its memory addressable range. Thus, the programmer can directly pass all customizing parameters to the *MC-BFP*. Each *SPR* is used for storing a system configuration parameter, a start/done flag, or a pointer to an external/internal memory entry. For this reason, we have divided the instructions into four different categories, based



(a) the RdSPR BF instuction (b) the BufCoef BF instruction

Fig. 6. BF instructions where the GPP reads from or there is no access to SPRs.

on the way the GPP accesses the *SPRs*. The categories are: *GPP reads SPR*, *GPP writes to SPR*, *GPP reads and writes to SPR*, *GPP does not access any SPR*, and are illustrated in Figure 6(a), Figure 8, Figure 9, and Figure 6(b) respectively. In each figure, a number highlights the corresponding step that is taken during the entire instruction execution. All instruction categories are analyzed next.

GPP reads SPR. As illustrated in Figure 6(a), *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 (step 1), it calculates the proper SPR memory address.

GPP writes to SPR. As depicted in Figure 8, *InStreamEn*, *ClrSPRs*, *DFirF*, *ConfC*, and *SSA* are the instructions that belong to this category. When the *InStream* instruction has to be executed, the GPP initiates a *GPP bus* write transaction and writes the *b_mask* value to SPR0 (step 1). Similarly, in *ClrSPRs* the GPP has to iterate through all *SPRs* and write the zero value to them (step 1). In *DFirF* instruction, the GPP uses the *Ftype* parameter to calculate the proper SPR address to write the *FSize* value (step 1). In *ConfC*, the GPP writes the *C* parameter to SPR9 (step 1), which is read from the partial reconfiguration controller, in order to load from the external memory the bitstream that includes *C beamformers*. finally, in the *SSA* instruction, the GPP iterates SPR[10+C] – SPR[9+2·C] and writes to them the on-chip *BF buffer* addresses (step 1), where 1024 input samples will be written, which are read from *buf_samp_addr*. Furthermore, it writes to SPR7 the *source buffer* address, where 1024 samples of the extracted source signal are stored (step 2).

GPP reads and writes to SPR. As illustrated in Figure 9, *LdCoef* and *BFSrc* instructions belong to this category. In *LdCoef*, the GPP writes all decimators coefficients addresses to SPR10 – SPR[9+C] (step 1), and the interpolator coefficients address to SPR8 (step 2), which are read from *buf_coef_addr*. As soon as all addresses are written to the proper *SPRs*, the GPP writes a *LdCoef start flag* to SPR4 (step 3) and remains blocked until the *MC-BFP* writes a *LdCoef done flag* to the same SPR. As soon as *LdCoef start flag* is written to SPR4, the *main controller* enables the *LdCoef controller* to start reloading the decimators coefficients (step 4). Once this step is finished, the *LdCoefInt controller* via the *main controller* initiates the interpolator coefficients reloading procedure (step 5). As soon as all coefficients are reloaded, the latter acknowledges the *main controller*, which writes a *LdCoef done flag* to SPR4 (step 6). This unblocks the GPP, which can continue further processing.

In *BFSrc*, based on the source aperture *aper*, the GPP calculates a *BF buffer* address offset, called *aperture address offset*, in order to access the proper H(z) coefficients sets. The GPP writes the *aperture address offset* to SPR5 (step 1). Furthermore, it performs a DMA transaction in order to read *C* 1024-sample chunks from the *xmem_read_addr* memory location and distribute them to on-chip *BF buffers* of the *C beamformer* modules (step 2). As soon as all data are stored, the GPP writes a *BFSrc start flag* to SPR6 (step 3). The MC-BFP reads the start flag from SPR6 (step 4), while the GPP

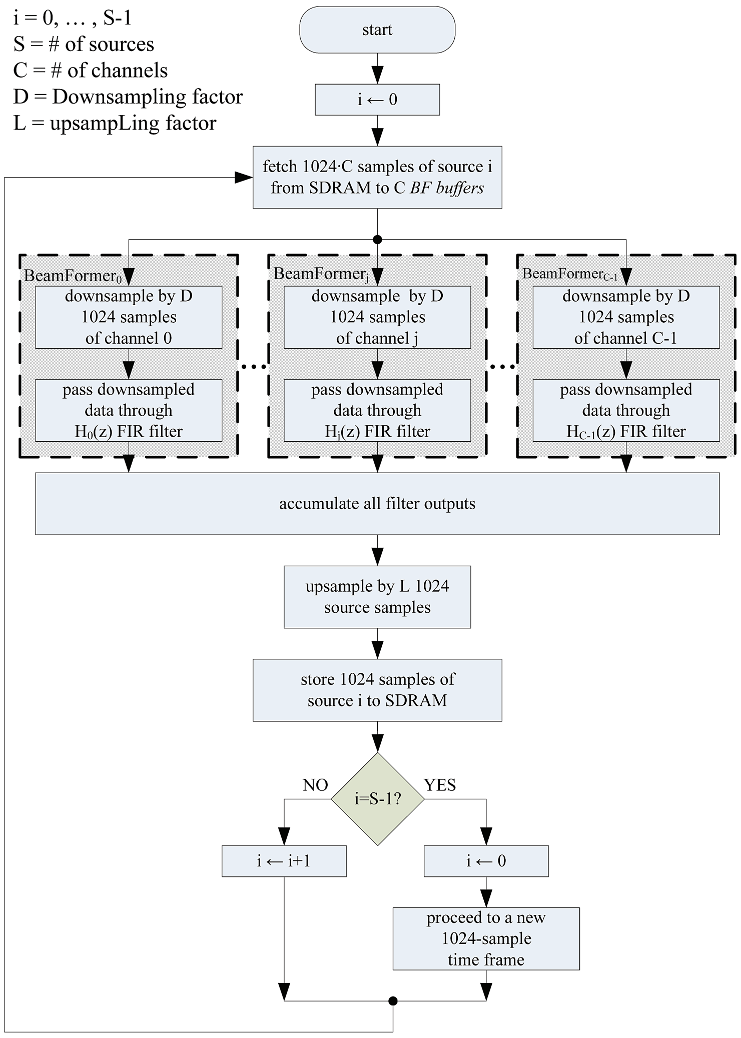

i = 0, … , S-1
S = # of sources
C = # of channels
D = Downsampling factor
L = upsampLing factor
start
i ← 0
fetch 1024·C samples of source i from SDRAM to C BF buffers
BeamFormer0
downsample by D 1024 samples of channel 0
pass downsampled data through H0(z) FIR filter
BeamFormerj
downsample by D 1024 samples of channel j
pass downsampled data through Hj(z) FIR filter
BeamFormerC-1
downsample by D 1024 samples of channel C-1
pass downsampled data through HC-1(z) FIR filter
accumulate all filter outputs
upsample by L 1024 source samples
store 1024 samples of source i to SDRAM
i=S-1?
NO
YES
i ← i+1
i ← 0
proceed to a new 1024-sample time frame

Fig. 7. Flowchart of the BF data processing among all RUs.

remains blocked until the MC-BFP writes a *BFSrc done flag* to the same *SPR*. Within each *beamformer* module, the *LdCoef controller* reads via *the main controller* the *aperture address offset* from SPR5 and reloads to the H(z) filter the proper coefficients set (step 5). Once all H(z) coefficients are reloaded, the *LdCoef controller* acknowledges the *BFSrc controller*, which enables processing of input data that are stored to the *BF buffers*. When all 1024 samples are processed, the *main controller* writes a *BFSrc done flag* to SPR6 (step 6), which unblocks the GPP. The latter performs again a DMA transaction, in order to transfer 1024 samples from the *source buffer* to the *xmem_write_addr* memory location (step 7).

GPP does not access any SPR. As illustrated in Figure 6(b), *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* arrays respectively. First, it performs a DMA transaction to transfer all decimator coefficients to the *BF buffers* (step 1). Next, based on the total number of source apertures to the system, it performs

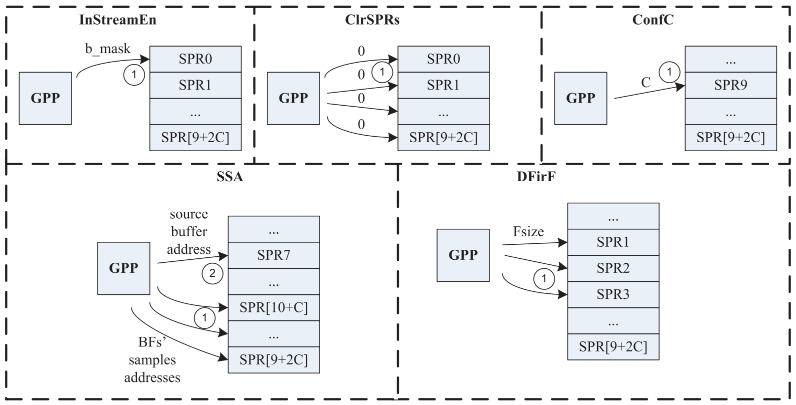


Fig. 8. BF instructions where the GPP writes to SPRs.

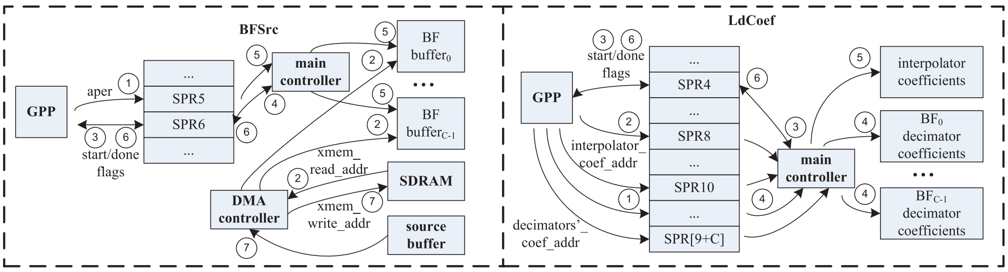


Fig. 9. BF instructions where the GPP reads and writes to SPRs.

a second DMA transaction to load all H(z) coefficients and distribute them accordingly to the on-chip *BF buffers* (step 2). Finally, with a third DMA transaction, the GPP fetches the active interpolator coefficients set to the on-chip *BF buffer* of *beamformer*$_0$ module.

5. EXPERIMENTAL RESULTS

FPGA prototype. We used the Xilinx ISE 9.2 and EDK 9.2 CAD tools to develop a VHDL hardware prototype of our MC-BFP. The latter was implemented on a Xilinx ML410 board with a Virtex4 FX60 FPGA and 256MB of DDR2 SDRAM. As host GPP processor, we used one of the two integrated PowerPC processors. Furthermore, we used the Processor Local Bus (PLB) to connect all peripherals, which are all on-chip *BF 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 when mapped onto a Virtex4 chip. When the design is mapped onto a Virtex6 FPGA, we utilize the Microblaze soft-core processor and the entire system is clocked at 200 MHz. Our prototype is configured with $C = 16$ *beamformer* modules, thus it can process up to 16 input channels concurrently. Also, within each BF-PE and the *source amplifier*, all decimators, H(z) filters, and the interpolator were generated with the Xilinx Core Generator.

Table III displays the resource utilization of each module when mapped onto Virtex4FX and Virtex6 FPGAs. The first two lines provide the required resources for a single *beamformer* and the *source amplifier* modules. The third line shows all hardware resources occupied by the MC-BFP. In the fourth line, we show the resources

Table III. Resource Utilization of Each Module when Mapped onto Virtex4 (V4) and Virtex6 (V6) FPGAs

Module	V4 Slices	V4 DSP Slices	V6 Slices	V6 DSP48E1 Slices	Memory(bytes)
Single BeamFormer	598	2	171	2	8192
Source Amplifier	2870	0	1086	0	2048
MC-BFP	14165	32	4824	33	133120
System infrastructure	6650	0	2080	0	317440
Complete system with C = 16	20815	32	6904	33	450560

Table IV. Maximum Number of *BeamFormers* that Can Fit in Different FPGAs

FPGA	# of BeamFormers fit
V4FX60	19
V4FX100	54
V4FX140	89
6VLX75T	78
6VLX760	360
6VSX315T	352
6VSX475T	532
6VHX250T	252
6VHX565T	432

required to implement the PLB, DMA, ICAP, and all memory controllers with their corresponding BRAMs. Finally, the fifth line provides all required resources from the entire BF system.

As can be observed, regarding the V4FX FPGA family, a single *beamformer* requires less than 600 slices, 2 DSP slices, and 8Kbytes of Block RAM (BRAM), which makes it feasible to integrate many such modules within a single chip. Table IV shows how many *beamformers* could fit into different V4FX FPGA chips. Moreover, even a medium-sized FPGA can support up to 19 channels, while larger chips, like the V4FX100 and V4FX140, could accommodate up to 54 and 89 input channels, respectively.

Of course, newer FPGA families, like the Xilinx Virtex6, integrate more resources, thus can fit many *beamformer* modules. In order to investigate how many input channels a single Virtex6 could accommodate, we used the Xilinx ISE 11.4 and mapped our MC-BFP onto different chips of the FPGA family. As we can observe from Table IV, a 6VLX75T FPGA chip, which is the smallest of the Virtex6 family, could fit up to 78 input channels. Moreover, the 6VSX475T chip, which is the largest one, could support setups that consist up to 532 microphones. We should note that during our calculations we took into account the required area to also map a Microblaze processor [Xilinx 2010a] onto the reconfigurable hardware, since the Virtex6 families do not integrate any hard-core processor.

Data accuracy. In order to evaluate the data accuracy of our FPGA implementation, we compared its output results against the ones from a GPP-based approach. As stimulation input, we used recorded signals from up to 16 microphones that consist of 24576 samples in a 16-bit signed format. However, the GPP-based BF application performs all calculations using the IEEE 754 standard single-precision floating-point format. In contrast, our MC-BFP follows a fixed-point format approach, in order to reduce area utilization and increase performance. However, due to the fixed-point format, a calculation error is introduced which results in a reduced output accuracy compared to the IEEE 754 floating-point format.

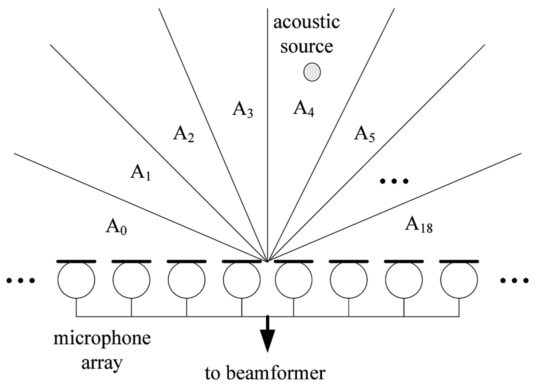


Fig. 10. Microphone array setup and source position inside aperture A_4.

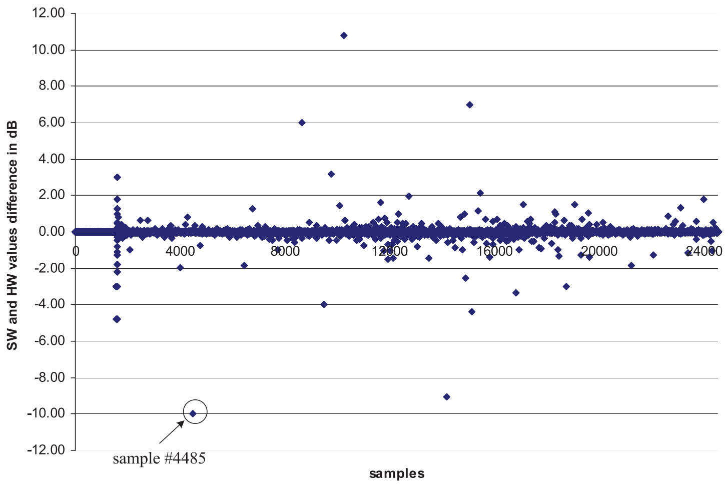


Fig. 11. Difference between software and hardware values for an acoustic source in dBs inside aperture A_4.

In order to verify that the reduced accuracy does not affect the extracted source quality, we compared the software and hardware sample values, in the exemplary case of a source being located within source aperture A_4 out of the total 19 currently available apertures, as illustrated in Figure 10. We used the following formula to estimate the introduced error for each calculated signal sample

$$DdB = 10 \cdot \log\left(\frac{s_{SW}}{s_{HW}}\right) dBs, \tag{5}$$

where s_{SW} and s_{HW} are the software and hardware sample values respectively. Figure 11 shows the introduced error for an extracted source signal consisting of 24576 16-bit samples. In the ideal case, the difference between the two values should be zero. As can be observed, almost all introduced errors do exceed a +/−0.01 decibels (dBs) boundary. In the few exceptional cases where the difference is large, it is because the absolute sample value is very low. For example, as depicted in Figure 11, the sample #4485 has a value difference of 10 dBs. This happens because the correct value s_{SW} is 1, however, the s_{HW} is 10. In practice, however, this would not introduce any loss in quality, since both values are very close to 0. In total, by taking into account these

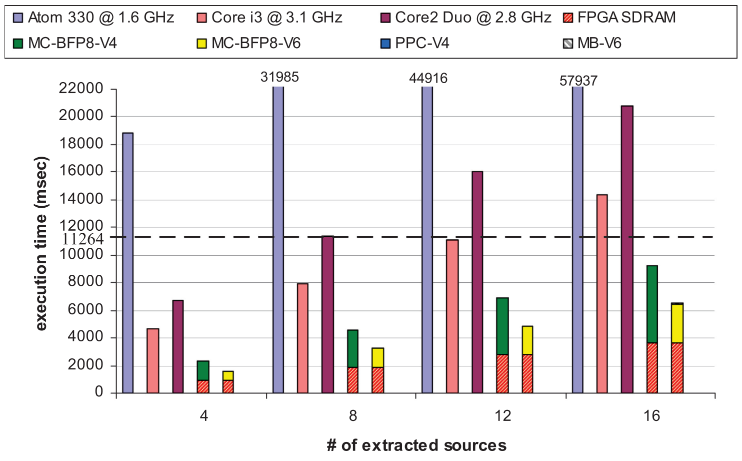


Fig. 12. Execution time on all platforms under an 8-microphone setup.

exceptional cases, we measured that the hardware output extracted source signal of our MC-BFP is 99.6% accurate to the software one.

Performance evaluation. In order to test the performance of all platforms, we provided input signals consisting of 540672 audio samples, divided into 528 1024-sample chunks. Assuming a sampling frequency of 48000 kHz, each iteration should be calculated in $\frac{1024}{48000} \frac{samples}{samples/sec} \approx 21.33$ msec. Since there are 528 iterations in total, every hardware platform that is considered for a real-time audio system must complete all data calculations within $528\ iterations \cdot 21.33 \frac{msec}{iteration} \approx 11.264$ sec.

To evaluate our approach, we conducted experiments for all hardware platforms with 8 and 16 channels and up to 16 sources. Regarding the different FPGA implementations, we use the "MC-BFPx-Vy" naming rule, where x defines the number of input channels the design uses and y refers to the utilized Xilinx Virtex FPGA family, that is y = 4 for Virtex4 and y = 6 for Virtex6. We compared both FPGA-based BF systems against an OpenMP-annotated software BF solution with SSE2 extensions enabled, which was implemented to three different GPPs, namely an Atom 330 running at 1.6 GHz, a Core i3 at 3.1 GHz, and a Core2 Duo at 2.8 GHz. The y-axis of Figure 13 depicts the achieved speedup by the FPGAs compared to all considered GPPs. As can be observed, the Virtex4-based MC-BFP with 8 *beamformer* modules (MC-BFP8-V4) and the Virtex6-based MC-BFP with 8 *beamformer* modules (MC-BFP8-V6) can process data up to 8.2 and 11.7 times faster compared to a low-power Atom 330 processor, up to 2.9 and 4.1 times faster compared to a middle-range Core2 Duo, and up to 2.0 and 2.9 times faster compared to a high-end Core i3.

The fact that the FPGA-based systems can process data faster compared to the GPPs obviously allows the support of more real-time sources. Figure 12 depicts the application execution times of each considered platform. As illustrated, the FPGA execution times are divided into the following parts:

—the "PPC-V4" part (for the Virtex4 implementations that use a PowerPC) or the "MB-V6" part (for the Virtex 6 implementations that use a Microblaze), which is the actual software execution time;
—the "FPGA SDRAM" part, which is the time spent on accessing the external memory;
—the actual data processing times of each FPGA platform.

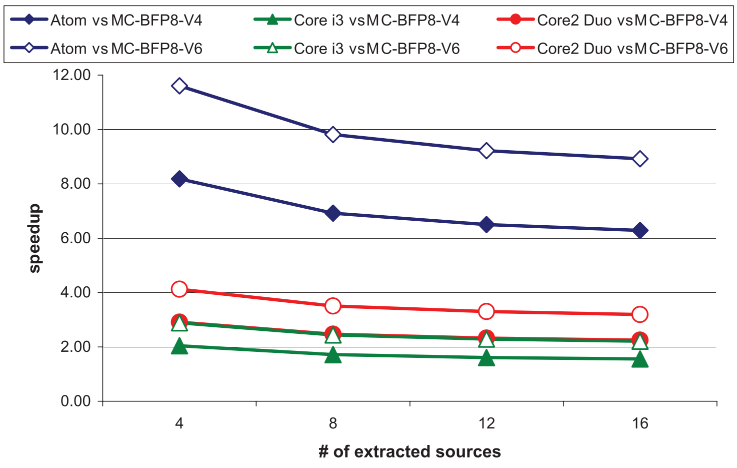


Fig. 13. Speedup against the GPP-based approaches under an 8-microphone setup.

It can be observed that the Core2 Duo and Core i3 implementations can process all data in real time, thus within 11.264 sec, for up to 8 and 12 sources respectively. In contrast, the Atom 330 processor could support extracting in real time only a single acoustic source. In case there are more sources required, the Atom processor failed to process all data within the required time limit. In the case of 12 and 16 sources, the Core2 Duo and Core i3 also fail to process data faster than the actual source length, thus making them not suitable for such real-time implementations. In contrast, the MC-BFP8-V4 and MC-BFP8-V6 systems could be used to build real-time BF systems that are capable of extracting up to 16 sources.

Figure 15 illustrates the achieved FPGAs speedup against the aforementioned GPPs when utilizing a 16-microphone setup. As can be observed, again both reconfigurable systems can process data faster compared to the GPPs. More precisely, the MC-BFP16-V4 and MC-BFP16-V6 systems can extract audio sources up to 10.6 and 13.7 times faster compared to the Atom, up to 3.7 and 4.8 times faster compared to the Core2 Duo, and up to 2.5 and 3.2 times faster compared to the Core i3.

Similarly to the case of the 8 input channels, the FPGA systems can process data more efficiently compared to the GPPs, thus they can support more real-time acoustic sources. In Figure 14, we provide the execution times of our experiments under a 16-microphone setup. Again, we have divided the FPGA execution times as explained before. As can be observed, the Atom-based BF system could not support real-time processing for any acoustic source. The Core2 Duo and Core i3 could be used for real-time BF systems when there are up to 3 and 6 sources to be extracted, respectively. In contrast, the MC-BFP16-V4 could be used for a BF system that supports up to 14 sources, while the MC-BFP16-V6 can support up to 16 sources.

As can be observed from Figure 12 and Figure 14, the software execution time on the FPGAs is very short and does not exceed 1.3% of the total execution time. In contrast, a significant processing bottleneck in both FPGA implementations is the external memory access times under both microphone scenarios. More precisely, under an 8-microphone array, up to 40% and 57% of the total MC-BFP8-V4 and MC-BFP8-V6 execution times, respectively, are spent on accessing the external memory. Furthermore, under a 16-microphone setup, the memory impact is increased up to 55% and 71% of the total MC-BFP16-V4 and MC-BFP16-V6 execution times respectively. The

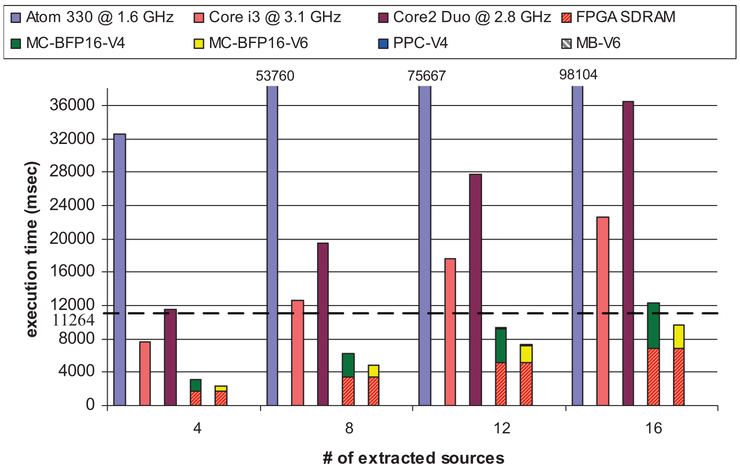


Fig. 14. Execution time on all platforms under a 16-microphone setup.

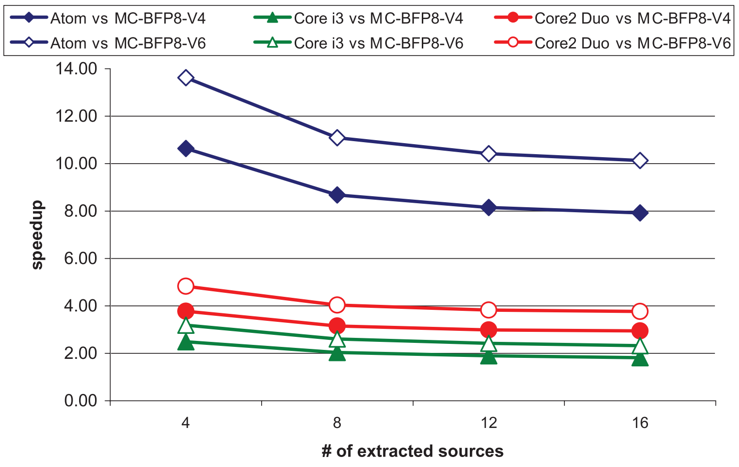


Fig. 15. Speedup against the GPP-based approaches under an 16-microphone setup.

reason that the external memory access time is relatively longer in the Virtex6 implementations than in the Virtex4 ones is because the former process data at 200 MHz instead of 100 MHz, however, both utilize the same external memory interface.

From the experiments conducted, we can draw the main conclusion that the external memory interface can significantly affect the performance of reconfigurable BF systems. As was suggested by Figure 4, in the current implementation we have utilized a single PLB to connect all peripherals, including the external memory controller. This approach adds a setup penalty time for each burst access mode, thus introducing an overhead to the memory accessing time. However, more efficient approaches can alleviate the slow external memory throughput and reduce the overall execution time. For example, a possible solution could be to remove the external memory controller from the common bus and develop a custom one that would connect all *beamformers* directly to a multibank off-chip memory. This way, each *beamformer* could load data in parallel from the external memory, thus reducing the memory load time when increasing the number of input channels. Furthermore, audio data prefetching to internal buffers of

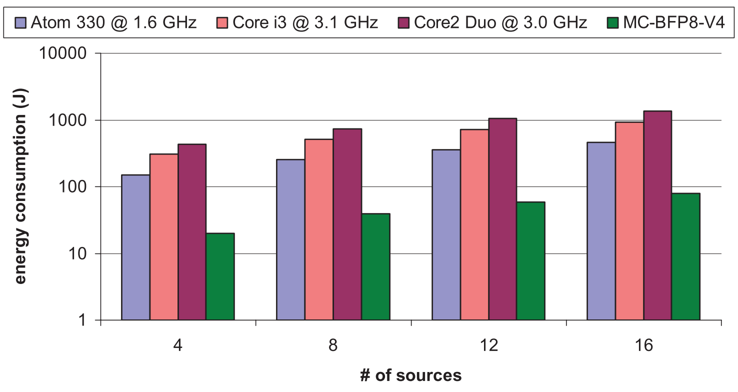


Fig. 16. Energy consumption of all processing platforms under an 8-microphone setup.

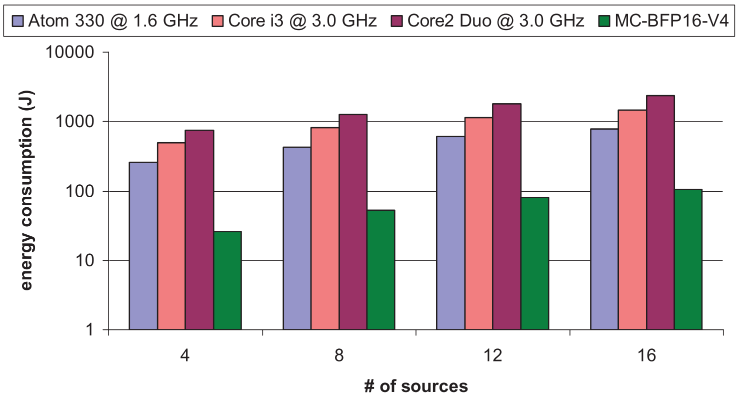


Fig. 17. Energy consumption of all processing platforms under a 16-microphone setup.

the next iteration while processing audio samples of the current iteration can reduce even more the memory impact on the overall execution time.

Energy consumption. Performance is not the only parameter to be considered when choosing a hardware platform for a BF system. Energy consumption is an additional parameter that should also be taken into account. Although the Atom 330 processor in most of the cases could not process all data within the required time limit, still it would be useful to provide its energy consumption, since it could be used with BF applications that do not require real-time processing. In order to evaluate the consumed energy from each platform, we used

$$E = P \cdot t, \tag{6}$$

where E is the energy consumed during the time t, while applying power P. The y-axis in Figure 16 and Figure 17 suggest the energy consumption by each processing platform under an 8- and 16-microphone setup for different number of sources. Due to the OpenMP annotations and the SSE2 extensions, the software execution utilized the GPPs approximately to 95%, thus we can safely assume that they were processing data at their peak power consumption. As can be observed, in every case the Core2 Duo-based system consumes the most energy. Even though its peak power consumption is 65 Watts [Intel Corp. Core 2 Duo], the fact that it requires more time to process all data compared to the MC-BFP16-V4 prototype results in excessive energy consumption. The Core i3 has also a peak power consumption of 65 Watts [Intel Corp. Core i3], however, it processes data faster compared to the Core2 Duo, thus consumes less energy. The Atom 330 processor has a peak power consumption of 8 Watts [Intel Corp. Atom 330], which is the least among all considered platforms. However, its slow data processing leads to an energy consumption that in many cases is an order of magnitude higher compared to the Virtex4-based approach. An interesting observation is that the Atom energy consumption is significantly less compared to the Core i3 and Core2 Duo due to its very low power. Finally, the MC-BFP16-V4 prototype, according to the Xilinx XPower utility [Xilinx 2010b], requires approximately 8.6 Watts, thus resulting in more than an order of magnitude less energy consumption compared to the Core2 Duo- and Core i3-based approaches. This is confirmed by the charts in Figure 16 and Figure 17.

Comparison against related work. Direct comparison against related work is difficult, since each system has its own design specifications. Moreover, to our best knowledge, we provide the first architectural proposal for reconfigurable BF. Previous proposals are mainly microarchitectural ones. In Yermeche et al. [2007], the authors utilize an ADSP21262 DSP, which consumes up to 250 mA. Furthermore, the voltage supply of ADSP21262 is 3.3 V [Analog Devices 2004], thus we can assume that the design requires approximately $3.3\ \text{V}\cdot 0.25\ \text{A} = 0.825\ \text{W}$. In addition, according to the paper, the ADSP21262 is 50% utilized when processing data from a two-microphones array at 48 KHz sampling rate, or alternatively 48000 samples/sec/input$\cdot 2$ inputs $= 96000$ samples/sec. Based on this, we can assume that 192000 samples/sec can be processed in real time with 100% processor utilization, which means $\lfloor 192000/48000 \rfloor = 4$ sources can be extracted in real time. Finally, in Yiu et al. [2008] the authors use four microphones to record sound and perform beamforming using an FPGA. They have mapped their design onto a V4SX55 FPGA and, according to the paper, every instance of the proposed beamformer can process 43463 samples/sec, with up to seven instances fitting into the largest V4SX FPGA family. Since the sampling frequency is 16 KHz, $\lfloor (43463\cdot 7)/16000 \rfloor = 19$ sources could be extracted in real time.

6. CONCLUSIONS

In this article, we presented a custom architecture for BF audio applications targeting FPGAs. The proposed architecture consists of 9 instructions, while the supporting programming paradigm employs a logically shared, physically distributed memory hierarchy. Our instructions allow customization and control of many system parameters, such as the number of input channels and filters size, while not requiring any low-level interaction with the hardware from the programmer. To evaluate our proposal, we implemented our custom architecture as a multicore reconfigurable processor and mapped it onto different FPGA families. We demonstrated through pseudocode that our approach combines the software flexibility of GPPs with the faster computational capabilities of the multicore platforms. Experimental results under different number of input channels and acoustic sources scenarios suggest that employing FPGAs for building the proposed BF audio systems leads to solutions that can extract in real time more acoustic sources compared to GPPs. Finally, based on the processing time and the power consumption of each considered platform, we evaluated their energy consumption and found that Xilinx Virtex4-based implementations require approximately an order of magnitude less energy compared to modern GPP-based BF systems.

REFERENCES

Acoustic Camera. 2013. http://www.acoustic-camera.com.

Analog Devices Inc. 2004. SHARC processor adsp-21262. http://www.analog.com/en/evaluation/21262-ezlite/eb.html.

BERACOECHEA, J., TORRES-GUIJARRO, S., GARCIA, L., AND CASAJUS-QUIROS, F. 2006. On building immersive audio applications using robust adaptive beamforming and joint audio-video source localization. *EURASIP J. Appl. Signal Process. 2006,* 196.

BERKHOUT, A., DE VRIES, D., AND VOGEL, P. 1993. Acoustic control by wave field synthesis. *J. Acoust. Soc. Amer. 93,* 2764–2778.

BUCHNER, H., SPORS, S., KELLERMANN, W., AND RABENSTEIN, R. 2002. Full-duplex communication systems using loudspeaker arrays and microphone arrays. In *Proceedings of the IEEE International Conference on Multimedia and Expo*. 509–512.

CEDRICK, R. 2005. Documentation of the microphone array mark III. http://www.nist.gov/smartspace/downloads/Microphone_Array_Mark_III.pdf.

CUTLER, R., RUI, Y., GUPTA, A., CADIZ, J., TASHEV, I., HE, L., COLBURN, A., ZHANG Z., LIU, Z., AND SILVERBERG, S. 2002. Distributed meetings: A meeting capture and broadcasting system. In *Proceedings of the International Conference on Multimedia*. 503–512.

FARINA, A., GLASGAL, R., ARMELLONI, E., AND TORGER, A. 2001. Ambiophonic principles for the recording and reproduction of surround. In *Proceedings of the 19th AES International Conference*.

FIALA, M., GREEN, D., AND ROTH, G. 2004. A panoramic video and acoustic beamforming sensor for videoconferencing. In *Proceedings of the IEEE International Conference on Haptic, Audio and Visual Environments and their Applications*. 47–52.

FILLINGER, A., DIDUCH, L., HAMCHI, I., DEGRE, S., AND STANFORD, V. 2007. NIST smart data flow system II: Speaker localization. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. 549–550.

THEILE, G. 2001. Multichannel natural music recording based on pychoacoustics principles. In *Proceedings of the 19th AES International Conference*.

INTEL CORPORATION. 2013. http://ark.intel.com/products/35641.

INTEL CORPORATION. 2013. http://ark.intel.com/products/36500.

INTEL CORPORATION. 2013. http://ark.intel.com/products/49020.

KAPRALOS, B., JENKIN, M., AND MILIOS, E. 2003. Audio-visual localization of multiple speakers in a video teleconferencing setting. *Int. J. Imaging Syst. Technol. 13,* 1, 95–105.

KYRIAKAKIS, C. 1998. Fundamental and technological limitations of immersive audio systems. *Proc. IEEE 86,* 941–951.

MEI, G., XU, R., LAO, D., AND KWAN, C. 2006. Real-time speaker verification with a microphone array. In *Proceedings of the International Conference on Pervasive Systems and Computing*.

MIHOV, S.G., GLEGHORN, T., AND TASHEV, I. 2008. Enhanced sound capture system for small devices. In *Proceedings of the International Conference of Information, Communication and Energy Systems*.

MIT CSAIL: MIT PROJECT OXYGEN. 2004. http://oxygen.lcs.mit.edu/.

MOUCHTARIS, A., REVELIOTIS, P., AND KYRIAKAKIS, C. 2000. Inverse of filter design for immersive audio rendering over loudspeakers. *IEEE Trans. Multimedia 2,* 77–87.

NILSEN, C.-I. C. AND HAFIZOVIC, I. 2009. Digital beamforming using a GPU. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. 609–612.

POLYCOM INC. 2009. Polycom CX5000 unified conference station. http://www.polycom.com/products-services/products-for-microsoft/lync-optimized/cx5000-unified-conference-station.html.

SALLBERG, B., SWARTLING, M., GRBIC, N., AND INGVAR C. 2006. Real-time implementation of a blind beamformer for subband speech enhancement using kurtosis maximization. In *Proceedings of the International Workshop on Acoustic Echo and Noise Control*. 485–489.

SNOW, W. 1955. Basic principles of stereophonic sound. *IRE Trans. Audio 2,* 42–53.

SQUAREHEAD TECHNOLOGY. 2013. Audio scope zoom audio. http://www.sqhead.com/.

TEUTSCH, H., SPORS, S., HERBORDT, W., KELLERMANN, W., AND RABENSTEIN, R. 2003. An integrated real-time system for immersive audio applications. In *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. 67–70.

TEXAS INSTRUMENTS INC. 2002. TMS320C62x/C67x power consumption summary. http://www.ti.com/lit/an/spra486c/spra486c.pdf.

THEODOROPOULOS, D., KUZMANOV, G., AND GAYDADJIEV, G. 2009. A reconfigurable beamformer for audio applications. In *Proceedings of the IEEE Symposium on Application Specific Processors*. 80–87.

THEODOROPOULOS, D., KUZMANOV, G., AND GAYDADJIEV, G. 2010. Minimalistic architecture for reconfigurable audio beamforming. In *Proceedings of the International Conference on Field-Programmable Technology*. 503–506.

VEEN, B. V. AND BUCKLEY, K. 1988. Beamforming: A versatile approach to spatial filtering. *IEEE ASSP Mag. 5,* 4–24.

WALL, K. AND LOCKWOOD, G. R. 2005. Modern implementation of a realtime 3d beamformer and scan converter system. In *Proceedings of the IEEE Ultrasonics Symposium*. 1400–1403.

WEINSTEIN, E., STEELE, K., AGARWAL, A., AND GLASS, J. 2004. LOUD: A 1020-node modular microphone array and beamformer for intelligent computing spaces. MIT/LCS Tech. memo MIT-LCS-TM-642.

XILINX INC. 2010a. The simple microblaze microcontroller concept. http://xilinx.eetrend.com/files-eetrend-xilinx/article/201105/1839-3354-xapp1141.pdf.

XILINX INC. 2010b. XPower estimator user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/ise_n_power_user_guide.htm.

XILINX INC. 2007. Implementing a real-time beamformer on an fpga platform. *XCell J.* 36–40. http://www.techonline.com/electrical-engineers/education-training/tech-papers/4137944/Implementing-a-Real-Time-Beamformer-on-an-FPGA-Platform.

YERMECHE, Z., SALLBERG, B., GRBIC, N., AND CLAESSON, I. 2007. Real-time implementation of a subband beamforming algorithm for dual microphone speech enhancement. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 353–356.

YIU, C. K., HO, C. H., LU, Y., SHI, X., AND LUK, W. 2008. Reconfigurable acceleration of microphone array algorithms for speech enhancement. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*. 203–208.

Received March 2011; revised February 2012; accepted May 2012