Technical Report

Design of Flexible Audio Processing Platforms using the System-on-Chip Environment

Wei-Cheng Su, Parisa Razaghi, Ashmita Sinha,
and Andreas Gerstlauer

UT-CERC-12-06

August 2, 2012

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967
http://www.cerc.utexas.edu

Design of Flexible Audio Processing Platforms using the System-on-Chip Environment

Wei-Cheng Su, Parisa Razaghi, Ashmita Sinha, Andreas Gerstlauer
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas, USA
{weicheng@, parisa.r@, gerstl@ece}.utexas.edu

Abstract—**This report demonstrates application of the System-on-Chip Environment (SCE) towards the design of hardware/software platforms for real-time audio processing following MP3, AAC, and AC3 decoding standards. Starting from C reference implementations, well-defined specification models are developed and fed into an SCE-based design space exploration and synthesis flow. Models are synthesized down to ARM-based HW/SW platforms, where in all cases, final software and hardware implementations are generated within minutes.**

Keywords-multi-processor system-on-chip (MPSoC); system compiler, audio processing

I. INTRODUCTION

Embedded systems and general purpose computing systems used to be at opposite ends of design spectrum, with little to no overlap between each other. While general purpose computing systems were powerful, flexible and programmable, embedded systems tended to be application specific, highly optimized and tightly constrained.

In the embedded world, we see today that more and more complex functionalities, such as HD video, web browsing or 3D graphics, are packed into embedded systems like smartphones. At the same time, increasing costs of chip development and shrinking time to market make it infeasible to design a new architecture for each new application instance. This has led to the emergence of platform-based design techniques, in which a more flexible, programmable or reconfigurable platform is reused across a large set of designs within a specific application domain. At the same time, while increasing the processing capability of systems is desired, cost, power consumption and heat are still the constraints that need to be dealt with. Pure software solutions have the most flexibility but consume too much power; while pure hardware solutions are more energy conservative but very inflexible. Hence, there is an inherent tradeoff between flexibility versus specialization that needs to be navigated.

In the general purpose computing field, the scaling of frequency has hit physical limits and power walls limit the growth of computing power using traditional programmable processors. Multi-core design has emerged as a solution to increase performance while limiting core clock frequencies and hence chip activity and power density. However, even when replicating the same existing cores multiple times on a chip, power concerns will limit achievable performance gains going forward. Hence, other accelerating components such as GPUs have become popular to offload the burden of CPUs towards more specialized and more efficient processing engines.

Overall, the development of embedded systems and general purpose systems both head toward solutions with multiple heterogeneous cores. Heterogeneous system design, however, is a complex task. Challenges include the high degree of parallelism at various levels, heterogeneity of programming models, architectures and tools, and ever-present real-time, power, cost and reliability constraints. Also, different parts of the system need to be specialized for different applications, using different development tools, design flows and incompatible interfaces. These challenges all together make it complex to develop a system from system specification to implementation.

In this project, we aim to research flexible yet low-power embedded hardware/software platforms for real-time audio processing. The objective is to explore and design suitable architectures that can support multiple audio codecs, but can do so in an optimal fashion as determined by performance, cost and power consumption metrics. Our initial focus in is on investigation of possible architectures for audio decoding following MP3, AAC and AC3 standards. In doing so, we aim to apply design methodologies, technologies and tool flows developed in our group, e.g. as realized by the System-on-Chip Environment (SCE) [1]. The long-term goal is to translate insights gained from this study into novel design space exploration algorithms that can automate the process of determining an optimal architecture for a class of applications, including the capability to design for flexibility in supporting future applications within that class.

A. SCE

SCE is a comprehensive development environment that takes a specification model, a set of element libraries and user input on design decisions, such as platform allocation and mapping to generate Transaction Level Models (TLMs) [2] at different abstraction levels for rapid and early virtual prototyping and design space exploration using simulation and validation of these generated models [3]. Backend hardware and software synthesis tools then take generated TLMs and synthesize C and RTL code to further realize the final software and hardware implementations using traditional low-level design flows. Besides the application itself, SCE also incorporates models of the underlying platform, such as Operating Systems (OSs) and hardware into the generated TLMs as well as final implementations so that the behavior of

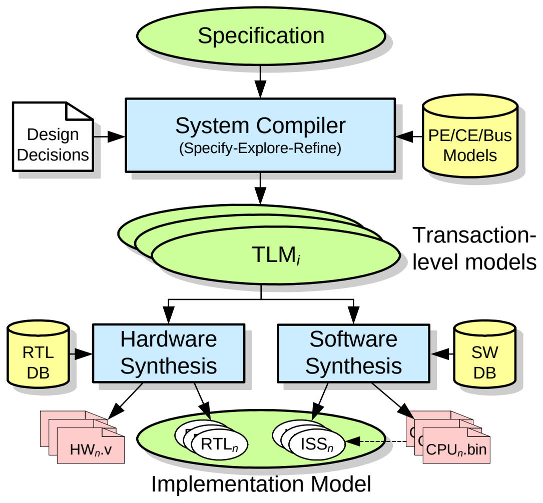


Figure 1: SCE design flow.

the whole system can be modeled, simulated and synthesized. In SCE, all models are all described in the SpecC System Level Design Language (SLDL) [4]. Final output is generated in C or VHDL/Verilog for software and hardware parts of the system. SCE has databases for processing elements (PEs), communication elements (CEs), buses, and operating systems. Figure 1 shows the design process using SCE.

SCE follows a specify-explore-refine flow when generating a lower level model. SCE takes the model generated from the upper level to *specify* the input to the design process at the next level. Designers enter different design decisions to SCE to generate models with different resource allocation, mapping, task partitioning, and scheduling. These models are then profiled and simulated to produce performance metrics such as power consumption and delay. The *explore* stage involves finding optimal solutions based on the metrics obtained. Finally, the optimal solutions are then *refined* into output models that are passed into the next level.

B. Audio Codecs

Audio codecs are used in many applications, such as music players, video and movie systems, cell phones, video/voice conferencing, to name a few. For different purposes, different categories of audio codecs are developed by different groups or standards. Audio codecs can be roughly divided into two categories: lossless and lossy. Lossless audio codecs encode all information, i.e. no information is lost after encoding and subsequent decoding. Examples of lossless audio codecs include the Free Lossless Audio Codec (FLAC), Adaptive Transform Acoustic Coding (ATRAC), WMA Lossless, etc. Generally speaking, these codecs provide the highest quality, although the quality also depends on the sampling rate and number of bits for quantization.

Lossy audio codecs, on the other hand, try to retain only significant information to reduce data rate while keeping a certain level of subjective quality. There are two types of lossy audio codecs. On the one hand, there are speech codecs, which analyze human voice content to find features in the encoder and synthesizes these features into equivalent speech in the decoder. Speech codecs are designed for encoding human voices and do not perform well with other type of sound like music. The most common speech coding scheme is Code Excited Linear

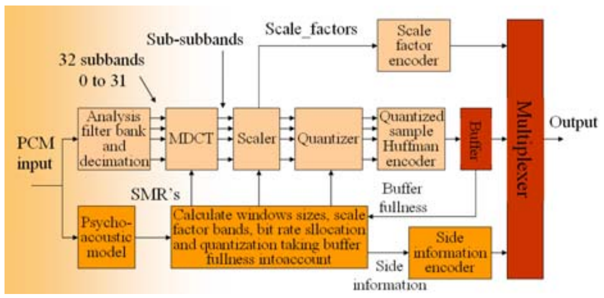


Figure 2: MP3 encoder block diagram.

Prediction (CELP). Examples of speech coding include G.723.1, G.726, iLBC, AMR, GSM 06.10, etc. Speech coding is mainly used for video/voice conferencing and cell phones. The other class of lossy audio codecs are psychoacoustic based. This type of codec removes the part of the audio that is unperceivable to humans based on psychoacoustic models. Compared to speech coding, psychoacoustic based audio codecs require higher bit rates and provide higher sound quality. They are widely used in many multimedia applications, such as music players, digital cameras, movie archiving, etc. Among the most popular and widely supported psychoacoustic audio codecs are MP3, AAC, and AC-3, which are especially popular in consumer electronics. Besides, these three codecs are open standards and many implementations are available. Thus, we choose these three codecs for design exploration within this project.

1) MP3

MPEG Audio Layer III, known as MP3, is a high-quality low bit-rate psychoacoustic model based audio codec. It works by removing the part of the audio that humans are not sensitive to in order to compress the size while maintaining high quality. It is defined in ISO/IEC 11172-3. MP3 is widely supported by almost all digital music players and is the most prevalent format used for music storage.

Figure 2 illustrates the block diagram of a MP3 encoder. Basically, the PCM samples are divided into subbands and the psychoacoustic model controls the configuration of MDCT (Modified Discrete Cosine Transform), Scaler, and Quantizer blocks to determine what information can be removed. On the decoder side, the procedure is reversed, but in a simpler way since a lot of psychoacoustic model computation is not required.

2) AAC

AAC (Advanced Audio Coding) is an improvement over MP3. It is defined in both MPEG-2 and MPEG-4. It is also a psychoacoustic based audio codec. Compared to MP3, AAC supports more bit rates, sampling rates, multiple-channel coding, and of course is more complex. shows the encoder block diagram of AAC.

Unlike MP3, AAC has several different profiles. Three AAC profiles are defined in MPEG-2. The LC (Low Complexity) profile is the simplest and the most widely used. The Main profile is like an LC profile with backward prediction. Finally, the SSR (Scalable Sample Rate) profile is designed to increase temporal resolution at high frequency and spectral resolution at low frequency.

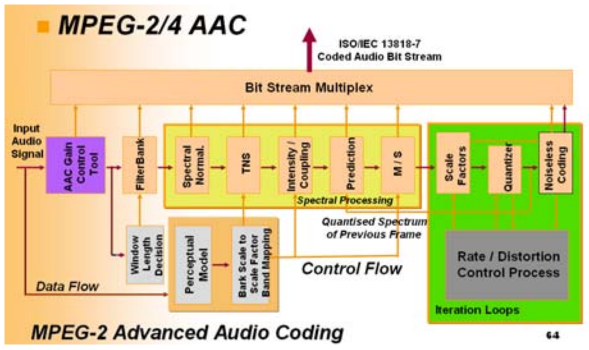


Figure 3: AAC encoder block diagram.

In MPEG-4, several audio profiles are defined and each profile includes several codecs. Two new AAC schemes are added in MPEG-4, i.e. HE-AAC (High Efficiency AAC) and LTP (Long Term Prediction). LTP is an improvement over the Main profile using a forward predictor. HE-AAC is an extension of LC-AAC optimized for low bit rate applications. HE-AAC v1 uses SBR (Spectral Band Replication), which takes advantage of harmonic frequency redundancy. HE-AAC v2 couples SBR with PS (Parametric Stereo) to enhance efficiency of stereo signals.

3) AC-3

AC-3 is also called Dolby Digital or ATSC A/52. It is commonly used to encode 5.1 channel audio, but also supports mono and stereo modes. Like MP3 and AAC, AC-3 is a psychoacoustic based audio coding technique. Figure 4 shows the block diagram of an AC-3 encoder and decoder. The filtered coefficients are represented with mantissa and exponents. Mantissa and exponent are transmitted separately. AC-3 takes advantage of a high frequency coupling that selectively couples channels at high frequencies

C. Audio Codec Implementations

To start the design process in SCE, a SpecC model is required. Since SpecC is very similar to C, it is comparatively easy to find an already available implementation in C or C++ and then convert it to SpecC. There are existing implementations provided by open source communities. When choosing the right open source implementation as a starting point for this project, we evaluated the following criteria: the coding language, fixed point versus floating point implementation, and code complexity. The following sections discuss the open source codecs we have surveyed and our final selection.

1) MP3

MAD [6] is a MPEG audio decoder that supports MPEG1 and MPEG2 formats. It implements all three layers – Layer 1, Layer 2, and Layer 3. MAD supports 24-bit PCM output and is implemented in C. Operations are done in fixed point. MAD is available under a GPL.

2) AAC

FAAD2 [10] is an open source MPEG-2 and MPEG-4 AAC decoder. It is licensed under GPL v2. It supports a variety of different AAC profiles such as LC (Low Complexity), Main, LTP (Long Term Prediction), and HE (High Efficiency). It is implemented in pure C code. The output format can be

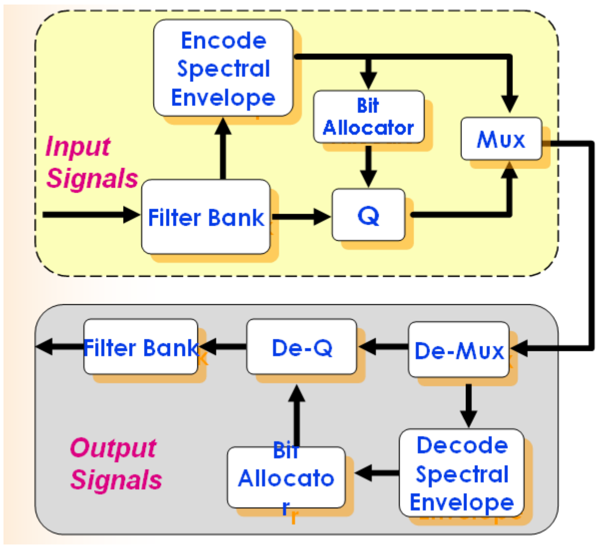


Figure 4: AC-3 block diagram [5].

configured to 16-bit, 24-bit, 32-bit fixed point integer or single or double precision floating point. However, the complexity of the source code is not low. The total number of lines of code is about 68,000.

The ISO AAC reference code [11] includes an encoder and decoder for all MPEG-4 audio codecs. The AAC decoder part has about 30,000 lines of code. However, the disadvantage of the reference code is that it is not optimized compared to other open source solutions.

Intel released an AAC code [12] that is especially optimized for IPPs (Integrated Performance Primitives). Though the code is well optimized with high performance to be expected when running on supported Intel platforms, tight coupling to a specific architecture may hinder our design space exploration.

Opencore AAC [13] is developed by PacketVideo and included as part of the Android multimedia framework. It supports LC, LTP, and HE profiles. It is mainly written in C, but is wrapped with C++ interfaces. The number of lines of code is about 75,000. Opencore also includes other audio codecs in the Android framework, such as MP3 and AMR.

Finally, the Helix AAC Decoder [9] is developed in open source form by RealNetworks. It is a 32-bit fixed point decoder and can be optionally optimized for ARM architectures. It supports the LC profile as well as the HE profile with SBR (Spectral Band Replication). Its code complexity is relatively low compared to other open source AAC codecs. The total number of lines of code is about 13,000. It is mainly written in C, but also has some code written in assembly for optimization on ARM.

Table 1 summarizes the features and capabilities of the above described open source AAC decoders.

3) AC3

Liba52 [8] is an open source implementation of AC-3 decoder under GPL. It is implemented in C and operations are done in floating point. The total number of lines of code is about 2791. Besides the decoding library, this package also includes a test program.

TABLE 1: SUMMARY OF OPEN SOURCE AAC DECODERS.

	Profiles	Lang.	LoC	Floating/Fixed
FAAD2	LC, Main, LTP, HE	C	~68k	Configurable for fixed point or floating point
ISO	LC, Main, SSR, LTP	C	~30k	Floating point
IPP	LC, LTP	C	~7.7k	Fixed point
Opencore	LC, LTP, HE	C, with C++ interface	~75k	Fixed point
Helix	LC, HE	C, ASM	~13k	Fixed point

D. Audio Code Selection

MAD was chosen as the basis for the design of the MP3 decoder because it is C based and 100% implemented in fixed point. Implementing such a design on a target processor avoids the need for a floating point co-processor. For the MP3 implementation, we leverage an existing SpecC-based MP3 decoder design based on the fixed-point MAD library as reported in [6].

For AAC, the Helix decoder was selected due to its relatively low complexity, 100% fixed point implementation, independence from other libraries, and being written in C.

For AC-3, Liba52 is the only codec surveyed and considered for implementation. This is based on an already partiallly converted SpecC model of Liba52 from a previous project.

II. AAC DECODER DESIGN

For the AAC decoder design, we start with the relatively simple fixed-point Helix reference implementation. The whole Helix project contains an audio codec framework that includes a couple of audio codecs. As a first step to prepare for conversion, the AAC decoder part was separated out into a standalone executable. This standalone AAC code was then further converted into a proper SpecC specification model for further design space exploration using SCE through a series of conversion and exploration steps as described in the following.

A. Specification

At the beginning of the conversion process, the C reference implementation was converted into an initial SpecC model. We first convert the C function call hierarchy directly into an equivalent hierarchy of SpecC behaviors, down to the level of primitive operations defined in the AAC source code that are kept as global or local C functions/operators. This initial SpecC code is “unclean” in the sense that the C function call hierarchy was converted directly into an equivalent hierarchy of SpecC behaviors that mimics the original C code (mapping each C function into one SpecC behavior) and mixes leaf statements with invocation of child behaviors throughout the hierarchy.

Next is to convert the “unclean” SpecC code into a proper SpecC behavioral and structural hierarchy that exposes available parallelism, accurately represents inherent dependencies, and cleanly separates computation and communication in each level. For this process, we replace pointers in communication interfaces with explicit arrays or channels. Global variables should also be removed and replaced by channels or local variables. Furthermore, we

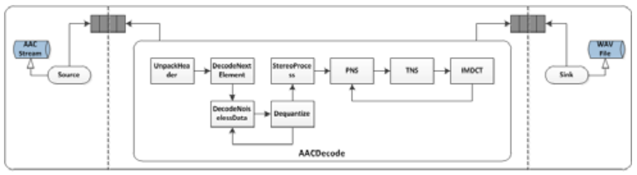


Figure 5: AAC decoder specification model.

reorganize the behavior hierarchy to cleanly separate C code in leaf behaviors from hierarchical instantiations of parent behaviors. For any non-leaf behaviors that mix regular C code statements with instantiations of sub-behaviors, we merge top-level statements into existing sub-behaviors or create new sub-behaviors. In the process, top-level control flow, like *if*, *while*, *for* statements are converted into an equivalent SpecC FSM composition. Furthermore, the code is simplified into a proper granularity for exploration by converting some smaller behaviors back into local functions and methods included in parent behaviors. This approach can lead to creating big leaf behaviors that may include many local methods. In the process of design space exploration, big behaviors can in turn be decomposed into smaller ones to increase flexibility if needed. Likewise, in future work, we may have to investigate further parallelization of the code to refactor FSM compositions into parallel or pipelined executions where possible.

Revisiting the hierarchical granularity was done to both decompose large behaviors into smaller ones and increase flexibility as well as to combine and merge several smaller behaviors (in some cases consisting of single lines of code only) across the hierarchy in order to reduce the structural overhead. Overall, a key aspect in developing a good specification model is the choice of the granularity of behaviors. For the purpose of synthesis and exploration, SCE considers behaviors as indivisible units of computation. As such, their hierarchical composition can significantly influence quality of results or complexity of the exploration process.

To validate the SpecC model, we implemented a proper test bench that is decomposed into separate modules for stimulus, monitor and the design under test (the actual AAC decoder). A stimulus behavior reads an input file and passes the content to the decoder via a queue. The design under test (decoder) decodes the incoming bit stream and outputs wave file data via a queue channel to the monitor when a frame is decoded. Another channel connects the decoder to the monitor for error reporting. A monitor behavior is added to receive decoded data from the decoder and write the output to a file. Stimulus and monitor modules terminate the simulation when either the end of the input file has been reached and completely decoded, or when a specified number of frames (as given on the command line) is reached.

In the process of developing the testbench and the decoder models, we added support to process AAC files in both ADIF and ADTS format. The current testbench only exercises the AAC decoder with an ATDS file, but the code is prepared to handle ADIF files as well (yet untested). We use a 2-channel, 44.1 kHz AAC stream that contains 861 frames to test the code. Throughout the successive conversion process we thereby ensure that the model is functionally accurate as validated against this fixed testbench, which is derived from test vectors provided with the reference code.

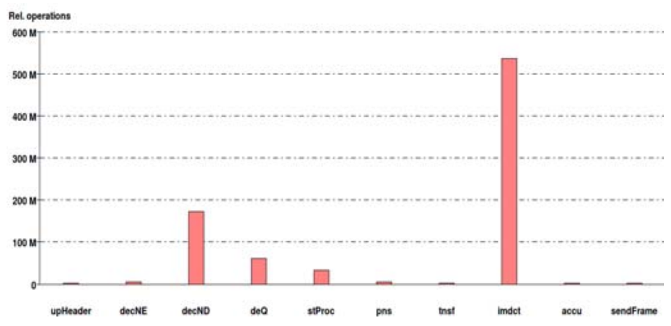


Figure 6: Computation profile of AAC decoder.

The block diagram of the final, complete AAC SpecC specification model is shown in Figure 5. The model contains 7227 lines of code, 22 behaviors, 16 leaf behaviors, and 3 channels. This SpecC model can be fed into SCE for design space exploration.

At this point, the conversion is complete. The SpecC specification model conforms both to the rules for use by the SCE tool set as well as to the modeling guidelines for proper granularity and encapsulation towards effective synthesis. Throughout all stages of the conversion process, the code has been functionally validated to produce output equivalent to the original reference code. The converted AAC decoder specification model can in turn be fed into the SCE tool flow for exploration and synthesis down to a variety of hardware and software implementations.

B. *Design Space Exploration*

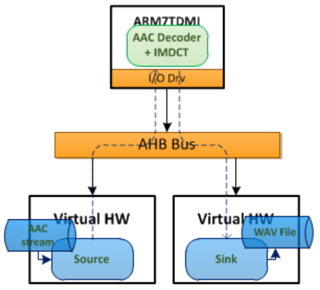
We start design space exploration by feeding the SpecC model into SCE. After compiling and simulating the input model, we profile the model and examine the computational requirements of each block. Profiling results are shown in Figure 6. It can be easily observed that the IMDCT block consumes the majority of the total computation. Thus, two possible architectures were adopted for this model. One is using a pure software solution, i.e. mapping all blocks onto a processor. The other is to add a custom piece of hardware that accelerates the most computationally intensive block, which is the IMDCT in this case.

1) *Pure Software Implementation*

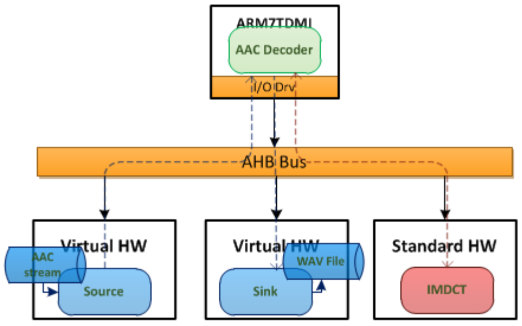
This implementation maps all blocks in this design to a single processor as illustrated in Figure 7(a). We select ARM_7TDMI among the processors provided by the SCE library as the only processing element and map all blocks under *AACDecode* to this processor. Two instantiations of virtual hardware blocks are allocated for stimulus and monitor I/O peripherals. For communication between modules, an AMBA AHB bus is allocated and the channels between stimulus and decoder and between decoder and monitor are mapped to the bus. The ARM processor is set as the master of the AMBA AHB and the two ports connecting stimulus and monitor are set as slaves. Each channel is given a different link layer address and interrupt.

2) *Hardware/Software Implementation*

To offload the most computationally intensive block from the processor, we allocate a custom hardware module in this design variant to accelerate this block. Figure 7(b) illustrates this implementation.



(a) Pure software implementation



(b) Hardware/software implementation

Figure 7: AAC design space explorations.

An ARM_7TDMI and a standard hardware PE are allocated for this design. The IMDCT block is mapped to the custom hardware accelerator and all other blocks under *AACDecode* are mapped to the processor. Similar to the pure software implementation, two pieces of virtual hardware are allocated for stimulus and monitor peripherals. Likewise, an AMBA AHB bus is instantiated for communication between processing elements. The processor is the bus master and the other three elements are bus slaves. Each processing element is assigned a range of addresses and an interrupts.

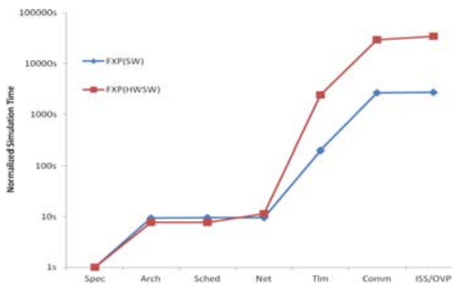
C. *Results*

Both design variants are synthesized down to the point of automatically generating the final target binaries for the ARM processors, whereas hardware models of IMDCT blocks remain at a behavioral level, i.e. are not yet converted down to RTL models following a high-level (C-to-RTL) synthesis process. The final synthesized implementations are validated by co-simulating binary code running on a functional, binary-translating (i.e. timing-accurate with CPI=1) instruction-set model of the ARM processor (using OVP [15] ISS models), which are embedded in an overall SpecC transaction-level and pin-accurate system simulation [16].

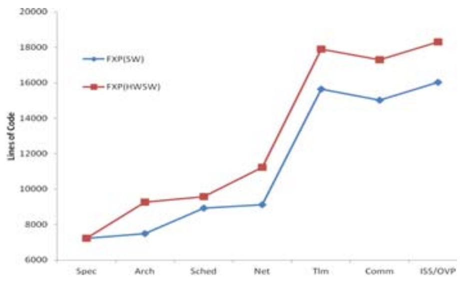
Table 2 shows the final decoding and simulation results of the two implementations as measured on the OVP-based TLM simulation. For each model, the average decoding delay per

TABLE 2: AAC DESIGN RESULTS ON OVP PLATFORM.

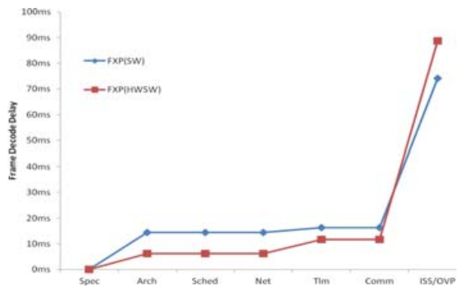
AAC Decoder	**SW-Only**	**HW/SW**
Avg. Decoding Delay	74.08 ms	88.74 ms
Simulated Instructions	3,346,299,772	3,853,005,626
Simulated Time	63.79 s	76.41 s
Refinement Run Time	88.191 s	88.079 s



(a) Simulation times



(b) Code complexity



(c) Simulated decoding delays

Figure 8: AAC decoder results.

frame and the total number of simulated instructions is provided. The simulation time is the time spent on the host machine to run the simulation for the input test file. Simulated time is the time needed by the design to decode the 861 frames of the testbench. The refinement run time is the time spent to generate the final implementation from the specification model.

Figure 8 shows normalized simulation time, lines of code, and simulated frame decode delay of both implementations at different levels of abstraction generated during the refinement process. As can be expected, with increasing detail included in the simulation at successively lower levels, simulation times and overall code complexities rise exponentially while simulation accuracy gradually improves towards the final ISS result. Note that simulated delays in models above ISS are based on back-annotated execution timings obtained from source-level profiling tools built into SCE [17], which are not well calibrated to the given target architecture. Accuracy of high-level models can be significantly improved by employing a fine-grain back-annotation of target-specific execution metrics [18]. Also note that due to the simple timing model in OVP, ISS results are not fully cycle-accurate either. This can lead to a misrepresentation of relative performance of different designs. Instead, as AC3 decoder results will show, high-level models are able to predict relative trends with better fidelity.

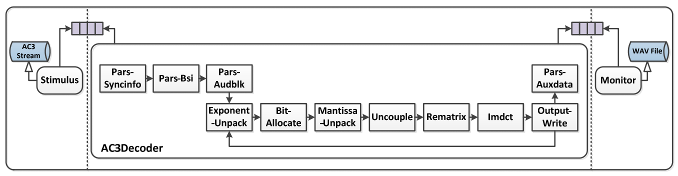


Figure 9: AC3 decoder specification model.

III. AC3 Decoder Design

For the AC3 decoder, we began with an existing, partial conversion of a floating-point C reference model into SpecC format that was performed in an earlier project. There were several issues with this initial model that needed to be resolved: (a) timing issues due to improper parallelization leading to deadlock situations, (b) unclean structural hierarchy with communication through global data structures instead of local variables and channels, and (c) use of floating-point instead of fixed-point arithmetic as a basis for an efficient embedded implementation. For the floating-point to fixed-point conversion, the goal is to perform conversion that can produce the same level of sound quality of the decoded outputs as compared to the floating-point reference implementation.

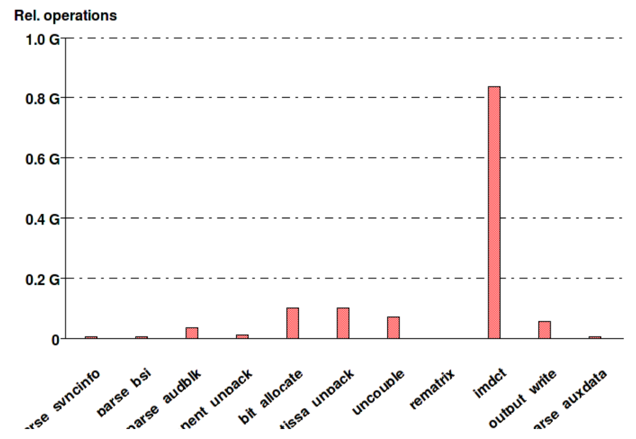
A. *Specification*

Following the same principles for converting the AAC model, the AC3 floating-point model was first converted into an unclean hierarchy and then further cleaned up, debugged and validated. The overall structure of the AC3 decoder SpecC specification model is shown in Figure 9. We have added a testbench into the model with capabilities for reading encoded AC3 files and writing the decoded output stream into a wav file (using the original reference code for downmixing of 5.1 audio streams into a stereo wav file representation). The model has been validated on several AC3 sample files with varying characteristics, e.g. in terms of sample rates, as obtained from the internet. In all cases, it produces bit-exact output when compared to the original reference code.

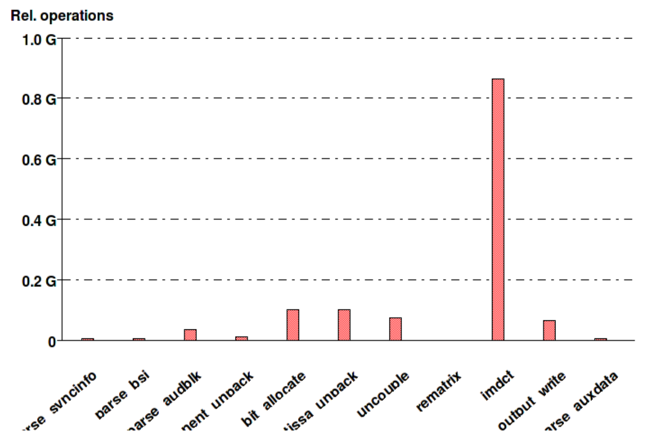
A fixed-point version of the AC3 decoder was also developed for this project. The original model contains four behaviors that include floating-point operations: Uncouple, Rematrix, Imdct, and OutputWrite (downmixing). These four behaviors are converted into equivalent fixed-point variants. The resulting fixed-point AC3 decoder specification model does not generate a bit-exact binary file. However, by listening to the generated wav files we could validate that floating-point and fixed-point versions produce output that is almost indistinguishable to the human ear. As such, we decide to not invest more time into increasing the precision of the fixed-point version. Rather, we concentrate on feeding both the floating-point and the fixed-point SpecC models into the SCE exploration and synthesis flow.

B. *Design Space Exploration*

To evaluate the performance of the AC3 decoder on various target architectures and to explore different configurations, we use the SCE framework to automatically generate target implementations from the SpecC specification models. We first performed profiling on the fixed-point and floating-point implementations. Figure 10 shows the profiling results for

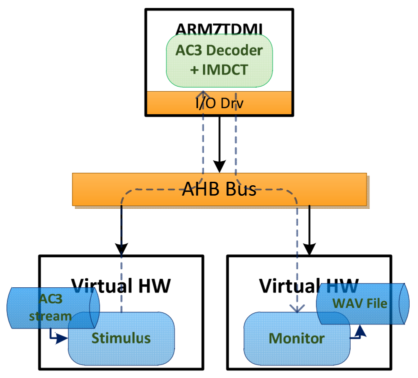


(a) Floating-point

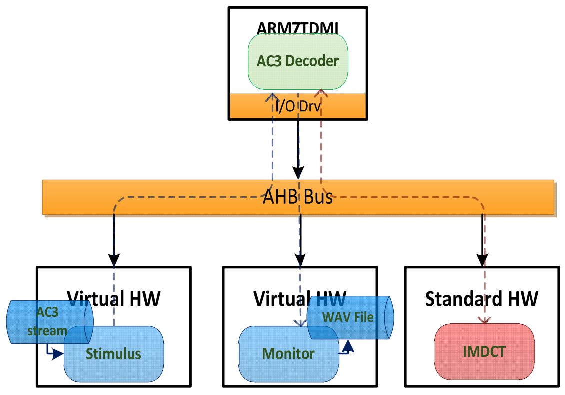


(b) Fixed-point

Figure 10: AC3 decoder profiling results



(a) Pure software realization



(b) Hardware/software design

Figure 11: AC3 decoder implementations.

floating-point and fixed-point implementations. As was the case in the AAC decoder, in both implementations, the IMDCT is the most computation consuming block in the system.

Thus, similar to the case in the AAC design, both fixed-point and floating-point models of the AC3 decoder were synthesized down to two different architectures: In a software-only architecture, as shown in Figure 11(a), all computation is performed on an ARM7TDMI processor running at 100MHz. In a hardware/software architecture, shown in Figure 11(b), the IMDCT component is mapped to a custom hardware block that sits on a common AMBA AHB system bus (running at 50MHz). In both cases, input and output blocks are realized by virtual hardware I/O peripherals sitting on the system bus.

C. *Results*

As before, all variants were synthesized down to the point of automatically generating the final target binaries for the ARM processors, whereas hardware models of IMDCT blocks remain at a behavioral/C level. In the AC3 case, the final synthesized implementations were validated by co-simulating binary code both on the timing-accurate-only OVP models as well as on a cycle-accurate instruction-set model of the ARM processor (using the SWARM simulator [14] embedded into a transaction-level or pin-accurate SpecC platform model).

Table 3, Table 4 and Figure 12 summarize performance and complexity for different architectures of the AC3 decoder when exercised with a testbench of 290 AC3 frames. The results show that better decoding performance is achieved in the fixed-point model, since the ARM processor does not contain a dedicated floating-point unit.

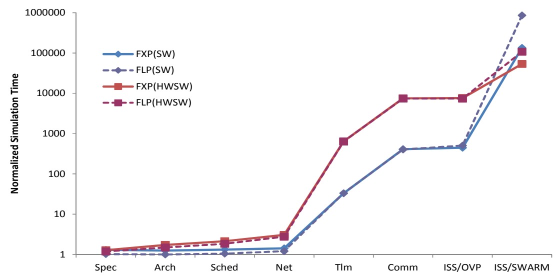
Furthermore, a large portion of floating-point operations is located in the IMDCT component, which leads to a significantly better performance in the HW/SW architecture of the floating-point model. By mapping the IMDCT and all of its floating-point operations to dedicated hardware, a better performance as for a fixed-point software implementation can be achieved. Nevertheless, hardware acceleration can even further improve fixed-point performance.

TABLE 4: AC3 FIXED-POINT RESULTS.

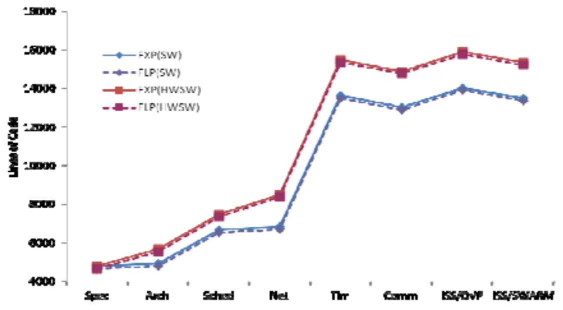
AC3Decoder	Fixed-Point Model	
	SW-Only	**HW/SW**
Avg. Decoding Delay	225.65 ms	80.92 ms
Simulated Instructions	1,479,629,294	1,264,151,548
Simulated Cycles	5,127,160,539	1,394,077,785
Simulated Time	64.44 s	23.47 s
Refinement Run Time	78 s	92 s

TABLE 4: AC3 FLOATING-POINT RESULTS.

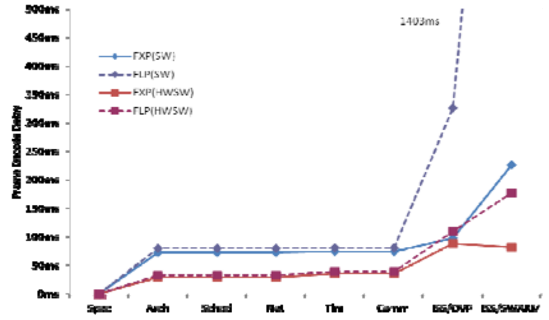
AC3Decoder	Floating-Point Model	
	SW-Only	**HW/SW**
Avg. Decoding Delay	1,401 ms	176.89 ms
Simulated Instructions	4,976,708,589	1,589,962,826
Simulated Cycles	32,340,289,561	3,528,134,535
Simulated Time	406.45 s	51.30 s
Refinement Run Time	77 s	91 s



(a) Simulation times



(b) Code complexity



(c) Simulated delays

Figure 12: AC3 decoder results.

Note that high-level models accurately predict the advantages of hardware acceleration in both floating- and fixed-point designs. However, the profiling tools used to annotate execution time estimates [17] do not yet reflect the lack of floating-point support in the chosen ARM processor. As such, simulated delays of floating- and fixed-point variants of each design are almost the same (small differences stem from variations in the size of floating and fixed-point data transferred between the ARM and other components).

IV. Conclusions

In this project, we have developed SpecC specification models for three audio decoding standards, namely MP3, AC3 and AAC decoders. All models follow a well-defined structure and model of computation (MoC) as a basis for further synthesis with the System-on-Chip Environment (SCE). With the given AC3, AAC and MP3 setup on top of the SCE tool set, we are able to quickly generate new implementation variants. For all decoder variants, we have explored several architectures and synthesized them down to final implementations on an ARM-based HW/SW platform. In all cases, final implementations of software and hardware running on the ARM-based target platforms could be synthesized within minutes. This can provide the basis for further projects to investigate and developing actual, novel multi-workload exploration methods that can synthesize target architectures optimized across a set of applications.

As a first step, this may involve using existing SpecC-based source-level profiling tools to capture code characteristics of various computational blocks in the three algorithms. Using this information, we can envision concepts and techniques to extract microarchitectural similarities needed to optimal support computations in all three applications on a common platform of heterogeneous processing elements. Likewise, we can explore commonalities in the hierarchical graph structures that represent the data and control flow across behavioral blocks in each application. This can lead to a notion of architectural flexibility that captures the types of application graphs (and operation characteristics in each node) that naturally map onto a given platform. The long-term goal is to develop such concepts into a theory and algorithms that will enable automated design space exploration and design for flexibility across a class of current and future applications.

Acknowledgments

This project was partially supported by Intel.

References

[1] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, D. Gajski, “System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design,” *EURASIP Journal on Embedded Systems (JES)*, vol. 2008, Article ID 647953, 2008.
[2] L. Cai, D. Gajski, “Transaction Level Modeling: An Overview,” In *Proceedings of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Oct. 2003.
[3] A. Gerstlauer, S. Chakravarty, M. Kathuria, P. Razaghi, “Abstract System-Level Models for Early Performance and Power Exploration,” In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Jan. 2012.
[4] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.
[5] W. Lee, Digital Perceptual Audio Compression Standards Slides.
[6] A. Gerstlauer, D. Shin, S. Abdi, P. Chandraiah, D. Gajski, “Design of a MP3 Decoder using the System-On-Chip Environment (SCE),” UC Irvine, Technical Report CECS-TR-07-05, Nov. 2007.
[7] Underbit Technologies Inc. MAD: MPEG audio decoder. http://www.underbit.com/products/mad.
[8] Liba52, http://liba52.sourceforge.net/
[9] Helix AAC Decoder https://datatype.helixcommunity.org/2005/aacfixptdec
[10] FAAD2: http://www.audiocoding.com/faad2.html
[11] ISO AAC Reference Software: http://wiki.multimedia.cx/index.php?title=AAC_Reference_Software
[12] Intel IPP Based AAC: http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/219967.htm
[13] Opencore AAC: http://source.android.com/source/download.html
[14] SWARM Software ARM Simulator: http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html
[15] Open Virtual Platforms (OVP): http://www.ovpworld.org
[16] P. S. Bomfim, A. Gerstlauer, “Integration of Virtual Platform Models into a System-Level Design Framework,” CERC, UT Austin, Technical Report UT-CERC-10-02, August 2010.
[17] L. Cai, A. Gerstlauer, D. Gajski, "Retargetable Profiling for Rapid, Early System-Level Design Space Exploration," In *Proceedings of the Design Automation Conference (DAC)*, San Diego, CA, Jun. 2004.
[18] A. Gerstlauer, S. Chakravarty, M. Kathuria, P. Razaghi, "Abstract System-Level Models for Early Performance and Power Exploration" In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Sydney, Australia, Jan. 2012.