A framework for automatic generation of audio processing applications on a dual-core system

Etienne Cornu, Tina Soltani and Julie Johnson

etienne_cornu@amis.com, tina_soltani@amis.com, julie_johnson@amis.com
AMI Semiconductor Canada Company, 611 Kumpf Drive, Waterloo, Ontario, N2V 1K8, Canada

Abstract

In this paper we describe an open framework for programming a dual-core system whose architecture is designed for ultra low-power audio processing applications. We show how the system's architecture leads to a model of computation, i.e. a formalism that describes the execution, scheduling and interaction between components in a system. We also show how applications can automatically be created from a set of components that are specifically architected for the dual-core system in terms of resource usage and performance using a set of complexity metrics associated with the model of computation.

1. Introduction

There are many applications for ultra low-power audio signal processing. They include encoding, decoding and general quality enhancement of the audio signal in mobile phones, speakerphones, wireless headsets and a variety of medical devices. Fixed-function products are widely used in these applications, but fully programmable audio processing systems are used increasingly often due to the need for flexibility. As these systems need to match fixed function products in terms of performance and cost, this flexibility is often associated with complex programming as their architecture is optimized for specific applications.

The audio processing system presented in this paper is an example of a dual-core system-on-a-chip (SoC) whose architecture is designed for ultra low-power audio signal processing applications. The signal processing system features dual digital signal processing units and one configurable DMA controller for sample management, plus a number of audio-specific mixed-signal components, such as analog-to-digital converters, digital-to-analog converters, filters, and pre-amplifiers.

Applications deployed on the audio processing system share many common signal processing blocks, such as FIR and IIR filters, time/frequency domain transformations and vector multiplication. Systems deployed on the audio processing system also share different algorithms (assembled from blocks), such as noise reduction, dynamic range compression, feedback suppression and echo cancellation. As companies develop these signal processing blocks and algorithms, we observe an increasing desire to expand the lifetime of these blocks and algorithms and therefore to re-use them in other applications. This creates the need for a framework that facilitates modularity and the development of applications from existing components (blocks and algorithms). This framework should also provide the basis for future tools that will be able to generate code from a high-level description.

Given the uniqueness of the audio processing system's architecture [1] and its parallel processing units, a formalism that describes the execution, scheduling and interaction between components had to be developed. The framework described in this paper was designed specifically for the audio processing system, as many of the existing concepts and tools, such as TI's eXpressDSP (see [2]), the Simulink Real-Time Workshop (from The MathWorks) ([3]) and Ptolemy (see [4]), do not directly address the parallelism issue. For example, they lack the complexity measures that allow the fine-tuning of software applications developed for the audio processing system. The framework not only specifies how programmers should develop components, but it also specifies a model of computation adapted to the parallel nature of the audio processing system. In addition, it provides a set of complexity measures that allow application developers to properly address trade-offs and thereby build complete applications that take maximum advantage of the resources available on the audio processing system.

This paper begins with a description of the architecture of the dual-core audio processing system. It continues in section 3 with a description of the issues that need to be addressed by the programming framework. In section 4 we present a programming model that forms the basis for this framework. In section 5, we present a tool that brings together system code, a number of algorithms and complexity measures in order to demonstrate the mechanics behind application development and automatic code generation. Finally, in section 6 we present a set of conclusions and describe possible future work that could lead to high-level programming tool specific to the audio processing system.

2. System architecture

The audio processing system, shown in Figure 1, is a DSP-based system optimized for low-power audio applications such as wireless headsets. The digital part of the audio processing system ([1]) consists of three major components: a weighted overlap-add (WOLA) filterbank coprocessor, a 16-bit fixed-point DSP core, and an input-output processor (IOP) that manages incoming and outgoing audio samples in a set of FIFOs. These three components run in parallel and communicate through shared memory. The parallel operation of these components allows for the implementation of complex signal processing algorithms with low system clock rates, low resource usage and low power consumption. The system is particularly efficient for subband processing in the frequency domain: the configurable WOLA coprocessor efficiently splits the fullband input signals into subbands, leaving the core free to perform the other algorithm calculations. The audio processing system also includes standard digital interfaces such as PCM, UART and I2S for communicating with other devices. The analog part of the audio processing system features two digital-to-analog converters and two analog-to-digital converters.

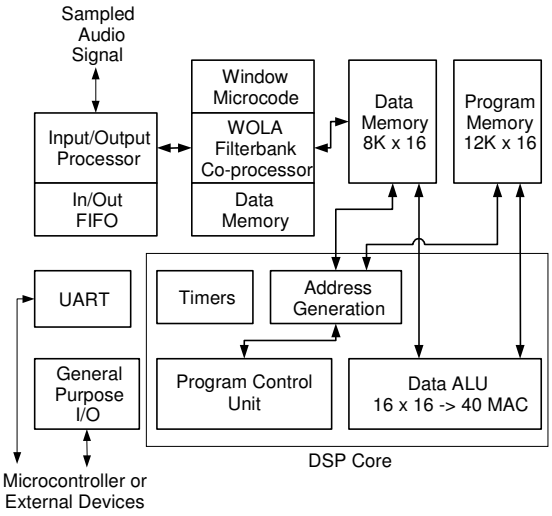

Figure 1 – Audio Processing System

The signal flows through the audio processing system as follows. After input samples are digitized by the analog front-end, the IOP stores them in the input FIFO. The IOP generates an interrupt on the core every R samples. The rate at which these interrupts occur is also known as the IOP tick rate. At every IOP tick, algorithms running on the core can perform operations such as IIR or FIR filtering on the time-domain samples stored in the input FIFO. When this is done, the WOLA coprocessor is launched. It processes the time-domain signal through an analyis filterbank and transforms the samples into the frequency-domain. When the analysis operation is complete, algorithms running on the core can perform operations on the frequency-domain data. The WOLA coprocessor is then launched again to apply frequency-domain gains and transform the data back into the time domain using a synthesis filterbank. Samples are stored in the output FIFO by the WOLA coprocessor. They are read by the IOP at every tick and sent to the digital outputs or converted to analog by the analog output stage.

3. Programming patterns

Although the system's architecture is designed for audio signal processing, the real benefits of this architecture can only be experienced when applications take full advantage of the system's features. These include efficient use of the system's parallelism, use of hardware support for efficient DSP calculations (e.g., the WOLA and built-in math functions), and frequency-domain processing approaches that utilize multi-rate techniques to reduce computation.

An analysis of a large number of blocks, algorithms and applications has provided important information regarding the development methodology and program structure typically employed on the dual-core system. The drive for efficiency means that no operating system is available to perform context switching. Also, developers write as much code as possible in parallel to minimize the clock frequency used (which saves power). To accomplish this, developers typically use an event-based approach where algorithms and background tasks are broken down into blocks, possibly in parallel, which execute in a deterministic number of cycles. The individual blocks of several different algorithms are usually integrated into one application in such a way that the algorithms that compose the application are sometimes not easily separable. Thus, a large portion of the development effort is spent optimizing signal processing blocks and algorithms to operate in this environment.

Given the cost of this optimization effort and the associated testing, re-useability is very important. In addition to being able to integrate individual components (blocks and algorithms) in different applications, different versions of a component that perform the same type of function but are optimized either for low power or for best performance can be made simultaneously available. The level of re-use also changes: sometimes source code is shared amongst integrators, and sometimes only object code can be shared because companies want to protect intellectual property. So it is important to be able to define re-useable components, to have information about the characteristics of these components available, and ultimately to be able to use these characteristics when a complete system is being built.

When the system components are chosen, a wide range of design and implementation trade-offs are then made on the complete system to obtain the lowest possible power

consumption and best possible functionality and performance. The number of configuration parameters in the system is relatively high. Parameters such as sampling frequency, number of frequency bands, band-to-channel mappings, filter lengths, and so on can affect the system in many different ways.

In summary, the following requirements for automatic application generation must be taken into consideration: (1) code shall be optimized for the specific dual-core architecture (taking into account parallelism), (2) modules will execute in a pre-determine number of cycles and have to be launched at precise moments, (3) different methods of performing the same function must be supported, (4) re-use at source or object code levels must be provided and (5), a large number of configuration parameters must be supported.

Our first step towards achieving automatic application generation was to define a software development standard that specifies all aspects of modular, re-useable software, including coding standards, interfacing standards, system-level behaviour, publication of component characteristics and a suitable model of computation. After this standard was defined and a number of algorithms were ported to comply with the standard, a tool for automatic application generation and characterization was developed. The software development standard and the tool are described in the following two sections.

4. Development standard

The result of the analysis of programming patterns is a software development standard ("Standard") for the audio processing system. This document specifies the structures and interfaces that are to be used when designing algorithms and establishes guidelines for the integration of these algorithms into applications. Three levels of interfaces are covered by the standard: the application or system level, the algorithm level and the signal processing block level.

In a typical algorithm, the WOLA filterbank coprocessor performs three functions every time a block of samples has been acquired: Analysis, Gain Application and Synthesis, all under the control of the DSP core which also performs its own tasks concurrently. These tasks can be grouped according to where they occur in relation to the WOLA filterbank's functions. These groupings are identified in Figure 2.

Accordingly, the Standard specifies that all code written for the audio processing system, excluding initialization code and interrupt service routines (ISRs), must be included in one of the following seven routines:

- *pre-analysis* – called after an IOP interrupt
- *while-analysis* – called after Analysis is launched
- *post-analysis* – called after an interrupt from the WOLA coprocessor indicates that Analysis has completed
- *while-gain* – called after Gain Application is launched
- *post-gain* – called after an interrupt from the WOLA coprocessor indicates that Gain Application has completed
- *while-synthesis* – called after Synthesis is launched
- *post-synthesis* - called after an interrupt from the WOLA coprocessor indicates that Synthesis has completed

This model of computation, related to Ptolemy's *discrete events* and *process networks* models of computation [4], promotes greater load balancing, parallelism and code readability when used in our context. It can also establish the relationship between a code block and the data it needs.

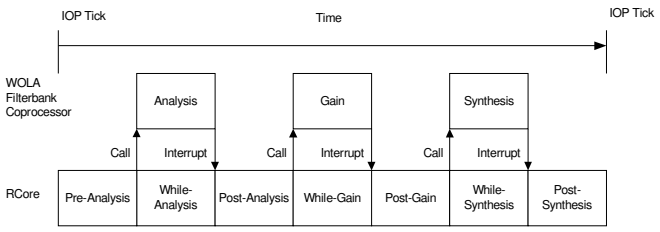

Figure 2 - Structure of a Typical Algorithm That Uses the WOLA.

In the Standard, algorithms are broken down into individual modules that will fit into these seven routines. Consider a typical algorithm operating on the Analysis results in order to calculate the gains to be applied during the coprocessor's gain application function; as such, a significant portion of the algorithm must be executed between Analysis and Gain. However, optimal parallelism is achieved if the majority of this algorithm is run in *while-analysis*, *while-gains* and *while-synthesis*. A typical method is to spread the gain calculation so that it is performed in *while-gains* and *while-synthesis* for the current sample block and in *while-analysis* for the next sample block. The resulting gains are then ready to be applied to the next sample block. This method can be applied when it does not noticeably impact the performance of the algorithm. Thus, the structure provided by the Standard has helped to clarify when the elements of the algorithm should be executed for maximum efficiency.

As well as signal processing operations, there are often system-level background processes that are not directly related to the signal path, such as power supply monitoring. The structure provided by the Standard allows the application integrator to see the critical path and thus schedule these background processes around it in such a way as to maximize

parallelism. The system integrator may therefore choose to perform background tasks in any of the functions *while-analysis*, *while-gains* and *while-synthesis*.

In addition to the topics described above, the development standard also addresses the following areas:

- Module interfaces
- Algorithm interfaces
- Memory usage and allocation
- Documentation requirements for modules and algorithms, including timing information
- General coding standards (including file structure, naming conventions, etc.).

The inclusion of each of these topics in the Standard results in code that is much easier to integrate. Although following the standard introduces a certain amount of overhead with respect to memory and cycle requirements, the benefits of following the Standard generally exceed such costs incurred. Depending on the system clock setting, the overhead in clock cycles can be as high as 20 percent. However, the overhead can be reduced by implementing shortcuts in the system code. For example, if no algorithm makes use of the *post-gain* function, the Synthesis operation can be launched directly when the Gain Application completes.

5. Automatic application generation tool

The purpose of the automatic application generation tool was to investigate and demonstrate the feasibility of automatically generating applications with coding efficiency that would rival hand-generated code. Besides code generation, the prototype tool also gives an indication of the application's complexity in terms of memory usage, parallelism and core cycle usage. As was identified during our analysis of programming and system development patterns, this information is absolutely necessary to allow users to fine-tune application and system parameters.

The operations involved in generating the application and complexity measures are described in Figure 3.

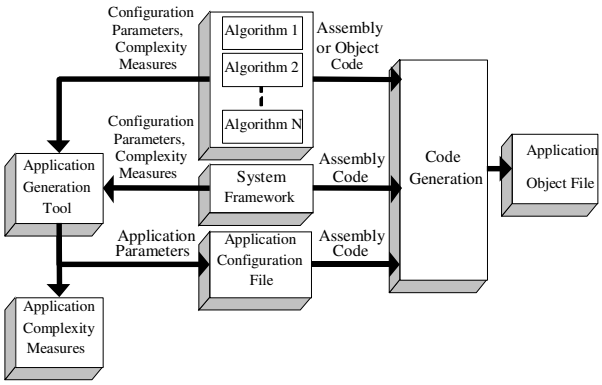

Figure 3 – Code generation data flow.

In order to be properly referenced by the automatic application generation tool, algorithms are implemented in accordance with the Standard as explained in Section 4. Each algorithm is divided into seven routines, based on the use of the data provided by the IOP and the WOLA coprocessor. These seven routines can be used by tool in assembly format or in object code for IP protection purposes. Complexity measures such as the number of clock cycles needed by each routine and their memory usage are obtained, and a number of configuration parameters are defined. They will be used by the tool to characterize and fine-tune the entire application.

A system-level module common to all applications performs the scheduling according to the defined model of computation. This module manages the interrupt signals generated by the three processors and calls the appropriate functions in the sequential manner illustrated in Figure 3. Complexity measures for this module are also determined.

After algorithms and system-level modules are developed and characterized, their configuration parameters and complexity measures are entered into the automatic application generation tool. The tool allows the user to select desired algorithms, their configuration parameters, and system settings such as sampling frequency and WOLA filterbank configuration. When all configuration parameters and settings are specified, a file that includes all of the information necessary to compile the source code for the desired application is generated. Memory usage for the entire application is also calculated by the tool.

As explained earlier, the optimal use of system resources is crucial in order to obtain the best possible performance from the application. The tool uses its knowledge of the audio processing system's dual-core architecture to determine how busy the main processor is expected to be when running the final application. This allows the application developer to investigate the feasibility of integrating certain algorithms based on the number of required clock cycles and the number of the available clock cycles.

The actual acoustic performance of the application can be immediately determined by compiling the algorithms, system code and application configuration file, and executing the resulting application object file on a real-time development platform. The process of selecting algorithms and configuration parameters, identifying resource usage, and testing on a real-time platform can then be repeated, as necessary.

6. Conclusions and future work

In this paper we have described a standard for the development of applications on the dual-core audio processing system. We have also described the prototype of an automatic application generation tool based on a coding

standard. The Standard and the tool will provide significant advantages for both algorithm developers and system integrators. For example, they will allow algorithms to be shared without the need to expose important intellectual property in the source code, and they will allow applications to be generated and fine-tuned in significantly less time.

Currently, the framework deals mostly with system integration at the algorithm level. Possible extensions in the future include adapting the framework to handle the integration at a lower level of abstraction, leading to re-usability at that level and, ultimately, to the ability to implement applications at a high-level while still retaining the efficiency of low-level coding techniques.

References

[1] R. Brennan and T. Schneider, “A Flexible Filterbank Structure for Extensive Signal Manipulations in Digital Hearing Aids”, Proc. IEEE Int. Symp. Circuits and Systems, pp. 569-572, 1998.

[2] Texas Instruments eXpressDSP Algorithm Standard Rules and Guidelines, SPRU352.

[3] The MathWorks, Inc. , SIMULINK Real-Time Workshop User’s Guide.

[4] Edward A. Lee, "Overview of the Ptolemy Project," Technical Memorandum UCB/ERL M03/25, July 2, 2003, University of California, Berkeley, CA, 94720, USA.