Modular SoC Integration with Subsystems

The Audio Subsystem Case

Pieter van der Wolf and Ruud Derwig
Synopsys
Eindhoven, The Netherlands
pieter.vanderwolf@synopsys.com, ruud.derwig@synopsys.com

***Abstract*-We explore the potential of subsystem-based design to reduce cost and time-to-market in the design of advanced Systems-on-Chips (SoCs) while retaining low-power and high performance processing. Using a concrete audio subsystem as an example, we illustrate the benefits of modular SoC integration with subsystems and identify challenges to be addressed. Well-designed subsystems pre-integrate hardware and software modules to implement complete system functions and offer high-level hardware and software interfaces for easy SoC integration. Configurability of subsystems enables reuse across SoCs. Subsystems can offer software plug-ins to support integration into a software stack on a host processor while making core crossings transparent for the application programmer. We conclude that subsystems can indeed be the next reuse paradigm for efficient SoC integration.**

Keywords— System-on-Chip (SoC); subsystem; audio;

I. Introduction

Modern devices like smartphones, tablets, and digital TVs are built using advanced Systems-on-Chips (SoCs) that integrate different system functions such as audio, video, modem, connectivity, imaging, etc. As the devices become more feature-rich, SoC integrators have to deal with increasing complexities and, thereby, increasing costs for the development of their hardware and software.

For cost and time-to-market reasons, SoC integrators purchase an ever larger fraction of the IP blocks for their SoCs from external IP suppliers. Purchasing commodity components helps SoC integrators to reduce the cost of ownership for in-house technologies, which require continuous effort to meet evolving requirements, and enables them to focus on their unique differentiators.

Moreover, in order to address complexity and facilitate multi-team multi-site development, SoCs are increasingly architected in a modular fashion as a set of coarse-grain subsystems for recognized system functions like audio, video, modem, etc. Such subsystems consist of multiple integrated hardware IP blocks, typically including one or more programmable cores (CPU, DSP), together with associated software. They typically implement complete system functions, with hardware and software interfaces for integration into the SoC. Subsystems are often developed by specialized teams having expertise of the targeted system functions and related implementation technologies.

Today, subsystem-based design is primarily practiced in-house by SoC integrators. They reuse subsystems across SoCs by having them evolve from one specific version to the next, as the requirements of the different SoCs have to be satisfied.

The next step to consider is an open market of subsystems for commodity system functions. Having subsystem integration performed by semiconductor IP suppliers for commodity system functions allows the subsystem integration effort to be amortized over multiple SoC integrators, and thereby increases efficiency in the semiconductor value chain. As such it is a logical next step in the evolution of the semiconductor industry after reuse of cell libraries and basic IP blocks.

In the context of SoCs for mobile phones, [1] identifies a trend towards a two-level architecture, in which functionally related cores, accelerators, and memory are clustered into well-defined subsystems, designed by specialists, each with its own life cycle, and potentially traded by competing suppliers. Subsystems as the next reuse paradigm, has also been emphasized in [2]. See Figure 1.

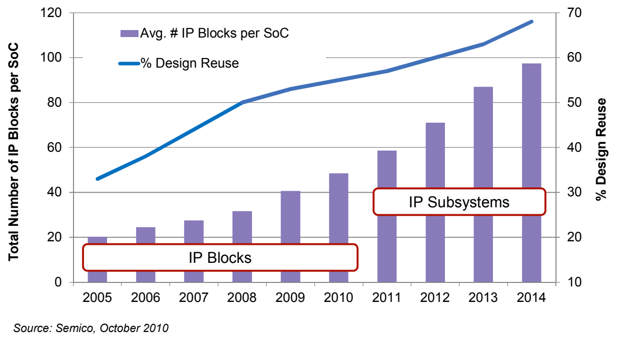

Figure 1. The trend towards pre-integrated IP subsystems

In this paper we explore the benefits of modular SoC integration with subsystems and identify challenges to be addressed, using a concrete audio subsystem as an example. In the next section we present the audio subsystem. In section III we generalize from the audio subsystem to other subsystems and discuss the conditions for making subsystem-based design, including the reuse of third party subsystems, an effective paradigm for modular SoC integration. We draw conclusions in section IV.

II. Example Audio Subsystem

A. *Audio Subsystem Functionality*

Audio processing is a key function in almost every consumer device. Prominent examples are digital TVs, set-top boxes, portable media players, tablets, and mobile phones. Integrating audio processing into SoCs for such applications can be a large and complicated task, particularly when different use cases and a variety of audio compression formats need to be supported. Specifically, this involves the integration of a range of hardware and software components, including an audio processor, audio peripherals, software drivers, and audio processing software. Today, SoC integrators typically build the audio subsystems for their SoCs themselves, using hardware and software components that are available on the market or developed in-house. This involves significant effort and risk.

In addition, new requirements for audio processing keep emerging. For example, there is a shift to multi-channel audio (e.g. 5.1 or 7.1) and higher sampling rates (up to 192 kHz). Advanced sound processing, such as virtual surround sound or automatic volume leveling, is performed to achieve a rich audio experience. More devices are equipped with voice features, and functions for voice pre- and post-processing, such as noise reduction and echo cancellation, are getting more advanced in order to achieve better speech quality. Also devices are getting more connected, demanding broader support for audio formats of the content that can be accessed.

An audio subsystem for a particular consumer device needs to support the range of audio use cases specified for that device. An example is a use case for playback of compressed Blu-ray Disc content, shown in Figure 2.

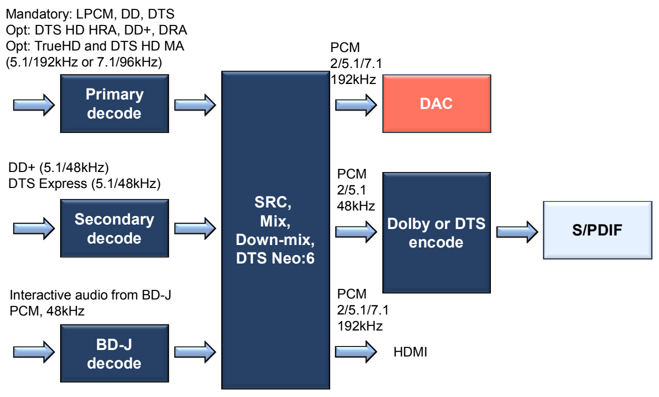

Figure 2. Blu-ray Disc playback use case

This use case has multiple input streams. The primary stream may be accompanied by a secondary audio track, with, for example, a film director's commentary. A third input stream with Blu-ray Disc Java (BD-J) content is used for system sounds for interactive menus, or BD-J controlled movie playback. The decoding functions must support a broad range of audio compression formats. Sample rate conversion converts all streams to the same (primary) sample rate for subsequent mixing. Audio encoding supports output of compressed streams over S/PDIF [8]. Uncompressed streams can be output via a DAC, for sound reproduction via speakers or headphones, or can be sent digitally over HDMI.

B. Audio Subsystem Hardware / Software Architecture

A key question upon the implementation of specified audio functionality in a SoC under design is whether the audio processing can be performed on the application processor or must be off-loaded to a separate audio processor.

A first criterion is the computational requirements of the audio processing functions in the use cases that are to be supported. These computational requirements can vary widely. Whereas MP3 decoding typically requires below 10 MHz to run on an audio DSP (measured with a memory access latency of 0 cycles), the Blu-ray Disc playback use case in Figure 2 typically requires several hundreds of MHz. Also for voice processing the computational requirements are increasing with the introduction of e.g. wideband voice codecs and multi-microphone beam forming. A RISC-style application processor without special provisions for audio processing typically requires a higher MHz budget for executing an audio processing function than a specialized audio DSP, which implements e.g. MAC operations for efficient execution of audio processing algorithms. Some application processors offer (optional) extensions to accelerate media processing, like the ARM Cortex-A series with the NEON extensions. Using these extensions the required MHz budget gets more in line with a specialized audio DSP. For example, employing NEON extensions, an application processor requires less than 10MHz to execute MP3 decoding [3][4]. Still, the MHz budget available for audio processing on the application processor in a particular SoC may not be sufficient, in particular when demanding audio use cases need to be accommodated.

In addition to resolving MHz budget issues on the application processor, off-loading audio processing to a separate audio processor can bring significant cost and power benefits. For example, a power-optimized ARM Cortex-A9 dual core with NEON extensions is 4.6mm^2 in a TSMC 40G process technology and uses a total power of 250mW per core when operating at 800MHz [5]. This brings the power consumption per core to 0.3125mW/MHz. In comparison, in the same process technology and at the same frequency, the power consumption (dynamic and leakage for both logic and memory) of an audio DSP like the Synopsys ARC AS211SFX audio processor [6] is about a factor 5 lower with an area that is a factor 6 smaller than a single Cortex-A9 core with NEON. Moreover, since for most SoCs the specified audio use cases do not require a budget of 800MHz, even higher power and area factors are achieved when the ARC AS211SFX audio processor is synthesized for a lower maximum frequency.

We conclude that off-loading of audio processing to a separate audio processor can resolve MHz budget issues on the application processor and brings significant savings in core power consumption, in particular for demanding use cases. In view of the increasing computational requirements of audio processing for many consumer devices, in this paper we consider the case where audio processing is off-loaded to a separate audio processor.

Moreover, a subsystem with strong internal cohesion and little external coupling that acts mostly autonomous, e.g. by including a local processor core and not depending on the application processor, eases integration by reducing interference and contention on shared resources. It helps meeting real-time deadlines and achieving low latency, both key quality parameters of audio processing in a system.

The diagram in Figure 3 presents an overview of the hardware / software architecture of the example audio subsystem which off-loads audio processing from the application processor.

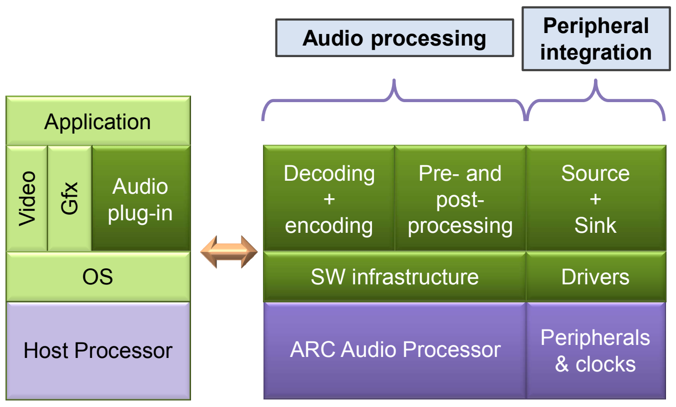

Figure 3. Overview of audio subsystem HW/SW architecture

The audio subsystem pre-integrates a range of hardware (dark purple) and software (dark green) components. The diagram also shows some of the hardware (light purple) and software (light green) components of a SoC in which the audio subsystem is integrated. Specifically, this is an application processor on which an application software stack is executed and from which the audio subsystem is controlled. Since the application processor hosts the audio subsystem, it is called Host Processor in this context.

The audio processing functions are executed on the ARC audio processor. These include codecs for decoding and encoding audio streams as well as functions for pre- and post-processing uncompressed audio streams. On the ARC audio processor, a software infrastructure consisting of a small Real-Time Operating System, a lightweight streaming framework, and an inter-processor communication (IPC) library supports the creation and execution of flow graphs of audio processing functions, controlled from the host processor. For a targeted use case, the SoC integrator can select audio processing functions from the portfolio supplied with the audio subsystem, or add his own if needed. Obviously, an audio subsystem adds most value if it offers a rich portfolio of audio processing functions so that all targeted use cases can be readily implemented. A further benefit offered by an audio subsystem can be that codecs have already been certified in collaboration with the standard holders where needed.

SoCs that provide audio functionality typically comprise dedicated peripherals for input / output of audio streams. Popular digital peripherals are I^2S [7] and S/PDIF [8]. An analog codec may also reside on the SoC to support analog inputs and outputs for microphone, line, headphone, etc. The audio subsystem includes these audio-specific peripherals, thereby providing a complete self-contained audio solution for easy integration into a SoC.

The module in the lower right of Figure 3 represents the hardware of the peripherals and a clock infrastructure, which we describe in more detail below. The software drivers for these peripherals are executing on the ARC audio processor in the audio subsystem. This simplifies the integration of the audio subsystem into a SoC, as the SoC integrator does not need to perform any software porting for peripheral integration. The input and output peripherals are made available to the application programmer as source and sink functions, which can be used as start- and end-points in use case flow graphs. The source and sink functions offer a high-level API, which hides details related to control of the peripherals and the clock infrastructure. All of this is handled transparently when source and sink functions are instantiated in a use case, started, stopped, etc. as facilitated by the software infrastructure.

The left side of Figure 3 illustrates the host interfacing of the audio subsystem. An audio plug-in is supplied with the audio subsystem to make the audio processing functions as well as the source and sink functions available on the host processor for building applications. The applications may combine audio processing with other desired processing such as video or graphics. The audio plug-in employs IPC between the host processor and the ARC audio processor.

The audio plug-in offers a high-level API for building and executing use cases. One of the supported APIs is the GStreamer API. GStreamer is a popular open-source multi-media framework for creating streaming media applications, and a large number of GStreamer plug-ins is available [9]. The audio plug-in makes all audio processing functions as well as the source and sink functions available on the host processor as GStreamer elements, which act as proxies for the actual functions on the ARC audio processor. Using the GStreamer API, SoC integrators can build and execute use cases as if they are executing locally on the host processor. This is illustrated in Figure 4 for a simple file playback use case. The File reader, Decoder and Renderer appear as local elements on the host processor. However, the audio plug-in transparently off-loads the Decoder and Renderer to the ARC audio processor. Furthermore, the software infrastructure takes care of all data streaming, i.e. both local and inter-processor communication, buffering, and synchronization of the processed audio data.

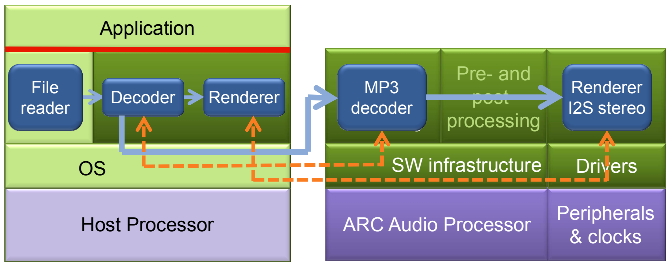

Figure 4. Use case controlled through high-level API on host processor

```
pipeline = gst_pipeline_new ("my-pipeline");
source = gst_element_factory_make ("filereader");
g_object_set (source, "location", filename);
g_object_set (source, "track", track);
g_object_get (source, "decodertype", &decodertype);
decoder = gst_element_factory_make ("decoder");
g_object_set (decoder, "decodertype", decodertype);
sink = gst_element_factory_make ("sink");
g_object_set (sink, "sinktype", I2S-STEREO);
gst_bin_add_many (pipeline, source, decoder, sink);
gst_element_link_many (source, decoder, sink);
gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

Figure 5. Abstracted code fragment for file playback use case

The abstracted code snippet in Figure 5 illustrates the simplicity of building the use case of Figure 4 from a set of GStreamer elements. It performs the playback of the track of a file referenced by 'filename' and 'track'. The audio format of the track is detected by the file reader and passed to the decoder. The use of the sink element of type I2S-STEREO realizes output of the audio stream over a stereo I^2S peripheral.

The audio plug-in greatly simplifies software integration while off-loading audio processing functions from the application processor to optimally achieve audio processing for demanding use cases. It provides a high-level, local, audio specific, functional API that hides implementation details about the off-loading such as cross-core streaming and control.

The hardware architecture of the audio subsystem is illustrated with the block diagram in Figure 6.

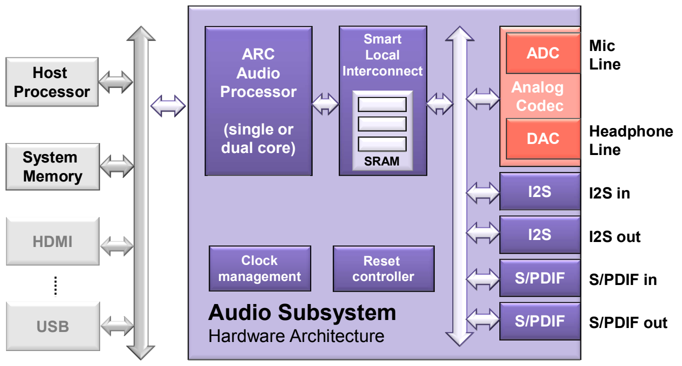

Figure 6. Hardware architecture of the audio subsystem

The audio subsystem hardware integrates the ARC audio processor and a set of audio peripherals using a smart local interconnect. Figure 6 shows a specific instance of the audio subsystem hardware with a particular set of peripherals. Actually, the set of peripherals as well as several other properties of the audio subsystem are not fixed but configurable. The reason for this is that the audio subsystem has been built to support reuse over different SoCs which can have quite different requirements on an audio subsystem. For example, the MHz budget of the audio processor required for a particular SoC strongly depends on the use cases to be supported by that SoC. Therefore configuration options have been implemented to allow the audio subsystem to be configured for use in a particular SoC. Three types of configuration options are supported:

1. Configuration of the number of audio processor cores as well as configuration of parameters of each core (such as cache size). The ARC audio processor can be configured as a single-core or as a dual-core audio processor, allowing the processing power to be scaled over a wide range in order to meet the demands of use cases.
2. Configuration of the number and type of peripherals (I^2S, S/PDIF; input, output; stereo, 5.1, 7.1; master, slave).
3. Configuration of properties that impact the integration of the audio subsystem in the SoC environment, like the type of bus interface (e.g. AHB or AXI) for interfacing to the SoC interconnect, base addresses in the memory map, etc.

Configurability spans both the hardware and the software. For example, the software needs to be configured for the selected set of peripherals.

The hardware architecture has been designed to provide an area-efficient solution that integrates easily into a SoC environment. The audio peripherals have been integrated with the ARC audio processor by means of a smart local interconnect, which provides an area-efficient infrastructure for audio streaming that meets the real-time constraints of the peripherals.

Each audio peripheral needs to be driven by a sample clock, either in master mode or in slave mode. If operated in master mode, the audio subsystem must supply a sample clock to the peripheral to drive the input / output of audio words. In order to simplify SoC integration, the audio subsystem includes a clock infrastructure to derive all the sample clocks for the configured set of peripherals from a source clock. A reset controller further facilitates SoC integration: resetting is done through a single "reset audio subsystem" command that hides the internal details of correctly resetting the individual components within the audio subsystem.

Note that audio data can enter / leave the audio subsystem also via the SoC infrastructure, e.g. when stored in system memory by other functions on the SoC such as a broadcast receiver or a file system.

We conclude that the audio subsystem pre-integrates hardware and software components to provide a configurable verified audio solution ready for plug-and-play SoC integration. Specifically it supports the SoC integrator by providing a solution for the following design tasks:

- Offering a rich suite of software codecs and pre-/post-processing functions on an audio processor
- Integrating digital and analog audio peripherals
- Interfacing to a host processor and offering a high-level API for building and executing use cases

The audio subsystem therefore helps to reduce cost and risk while accelerating time-to-market.

III. Subsystem Benefits and Challenges

Modular SoC integration with self-contained subsystems that offer high-level interfaces leads to more distributed SoC architectures, thereby enabling a divide-and-conquer approach in the design of complex SoCs. The audio subsystem exemplifies this in several ways. Drivers execute on the audio processor rather than on the application processor. Scheduling of tasks on the audio processor is performed locally. Sample clocks for the peripherals are derived locally from a source clock based on the sample rate of the audio stream, rather than in a central SoC-level clock factory that is controlled from the application processor. A local reset controller resets the components of the audio subsystem. The smart local interconnect and the local execution of drivers manage peripheral input / output so that real-time constraints are met. The increased autonomy of the audio subsystem limits dependence on the application processor and the software it executes, thereby simplifying integration.

The example audio subsystem demonstrates that modular SoC integration with subsystems can bring significant benefits.

With this example in place we can now investigate whether these benefits also apply to other subsystems and identify further challenges.

A. Standard Functions and Openness

The audio subsystem brings value to the SoC integrator by offering a rich suite of audio processing functions. Many of these functions are standard functions that offer little room for the SoC integrator to differentiate by having a dedicated implementation. This specifically holds for the broad set of software codecs that need to support standard audio formats like MP3, Dolby Digital (AC-3), WMA, AAC, etc. These are mainly judged by the efficiency of their implementation, which of course must be competitive for the audio subsystem. Audio pre- and post-processing can be more differentiating by employing advanced algorithms that yield a better sound quality. Therefore the audio subsystem allows the SoC integrator to add his own audio processing functions.

If we consider subsystems with high reuse potential for other domains, then one criterion is whether many functions in that domain are seen as standard, with little room for algorithmic differentiation. A subsystem for that domain can then bring value to multiple SoC integrators by offering a suite of efficiently implemented standard functions, and can be traded as third party subsystem IP on the open market. The domains of video decoding / encoding, graphics rendering, and multi-standard modem are likely candidates.

Other domains like video post-processing for picture quality enhancement, imaging and embedded vision do not yet have a widely agreed set of standard functions. Subsystems for these domains therefore need to be open subsystems that provide support for the development of new functions by the SoC integrator, if they want to be reusable. Such open subsystems can still offer significant value by addressing other hardware and software integration aspects. For example, they can offer an interface to a host processor and integrate peripherals specific for their domain.

B. High-level, Standard Interfaces

The audio subsystem was designed to be largely self-contained. High-level interfaces are used in order to realize loose coupling. For example, the reset logic for all IPs inside the audio subsystem is not exposed on the outside; the interface is only a single reset line. On the software side, the audio subsystem offers a plug-in with the high-level GStreamer API for building and executing use cases on the host. For decoding it provides a single, multi-standard audio decoder interface that hides the details of instantiating the specific audio decoder on the audio processor. Although the "loose coupling" heuristic for modular system design is well known, not all system functions are equally well suited for realization as a self-contained subsystem. Or, implementing a self-contained high-level interface could introduce unacceptable inefficiencies on system level. Careful system partitioning and interface design therefore is a key quality aspect of subsystems and remains a challenge.

The GStreamer API is especially popular in home equipment like Digital TV receivers and Blu-ray Disc players. However, SoC integrators also use other APIs, like the OpenMAX [10] standard that has been adopted in a number of mobile phones. Also, many SoC integrators rely on proprietary, in-house developed APIs. The lack of a single standard hampers reuse of the subsystem in an open market. Clearly, a single standard high-level API for the development of software applications on the host processor enhances the reuse potential of subsystems. In practice, however, there are very few "single standards" – certainly not in the highly fragmented embedded market space.

Fortunately, even more important than an exact match of the subsystem provided and SoC required interfaces, is the architectural match between subsystem and integrated SoC. As originally stated by Garlan e.a. in [11] and reconfirmed in [12], easy integration of reusable components is mostly hampered by "architectural mismatch" caused by assumptions on 1) the nature of components, 2) the nature of the connectors, 3) the global architectural structure, and 4) the construction process. If on the level of protocol, data, and control model the interfaces match, then specific differences can be relatively easily bridged by applying wrappers and glue logic. Using architecture specialization – like the dataflow paradigm for the audio processing domain, or by using open source practices – as the GStreamer framework, the likelihood of architectural mismatch is reduced.

C. Configurability

In section II.B we have shown that configurability is a key ingredient for achieving reusability of the audio subsystem. The diagram in Figure 7 puts the configurable subsystem approach in perspective by relating it to other approaches for implementing system functions in a SoC. It positions the various approaches in terms of the flexibility to adjust to specific requirements of a SoC and the ease of use for building and integrating the system function. Ease of use is inversely proportional to the engineering effort.

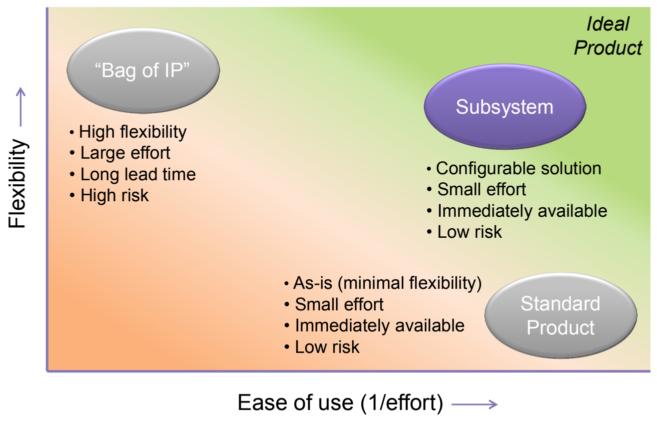

Figure 7. Configurable subsystem matches flexibility with ease of use

One approach is to define a fixed "standard product" and reuse that across multiple SoCs. For the audio domain this would imply an audio solution with a fixed audio processor and a fixed set of peripherals. However, as we have discussed, the audio solutions for different SoC applications may differ significantly. This also holds for other system functions. For example, for video decoding and encoding the requirements differ significantly depending on e.g. the frame size and the frame rate to be supported in a SoC. A one-size-fits-all standard product would either be too limited or occupy too much silicon area for use in a SoC.

The "Bag of IP" approach is the traditional approach towards implementing system functions for a SoC. Starting from the requirements of a SoC under design, the SoC integrator shops around for hardware and software IP and integrates these in the SoC. This approach offers a lot of flexibility but requires a large design effort, has a long lead time and involves significant risk.

A configurable subsystem matches flexibility with ease of use. It is readily available and can be integrated with little effort. Since the subsystem has been pre-verified, risk is significantly reduced.

Subsystem configuration is typically performed by the SoC integrator, driven by the requirements of the SoC under design. The set of configuration options of a subsystem needs to be carefully balanced with the ease of use. Offering too many configuration options may expose complexity to the SoC integrator, resulting in a larger effort in integrating the subsystem. Referring to Figure 7, adding configuration options may move the subsystem "up" since flexibility increases, but may also move it "to the left", which is undesired. Preferably, configuration options are limited as long as they do not render the subsystem too inflexible.

A configurable subsystem offers additional benefits that cannot be realized with a "Bag of IP" approach to the integration of system functions. A subsystem can simplify integration through streamlined tool support with a smart user interface and an automated configuration flow to generate the selected hardware and software configurations. By assembling all of the components into a subsystem, the entire block can be verified by the subsystem supplier in much the same way that a standard IP block is today. This relieves the SoC integrator of the burden of cobbling together and verifying components from multiple sources.

D. *Verification*

The subsystem is pre-verified by the subsystem supplier. This includes the verification of all possible configurations, which is another reason to carefully choose the set of configuration options. Verification starts with the verification of the basic hardware IP blocks. This needs to be performed only for the configurations of the IP blocks used in the subsystem and to the extent to which the IP blocks can be exercised by the software in the subsystem. For example, the peripheral drivers use the peripherals in a specific way. Subsequently, subsystem hardware verification performs integration verification using extensive test-benches while traversing the subsystem configuration space. Hardware / software verification is driven by module-level software testing and an extensive set of example use cases and associated regression tests.

The configurability increases the verification effort for the subsystem supplier. However, since the subsystem supplier knows the integration context of the hardware and software components, he can also significantly prune the verification space and thereby reduce effort compared to supplying separate IPs that need to be extensively verified without knowledge of the (sub)system integration context. For the SoC integrator the verification effort is reduced as the integration verification performed by the subsystem supplier does not have to be redone.

E. *Prototyping*

Having a pre-integrated subsystem also opens the opportunity to support SoC integration with prototyping solutions. These solutions can consist of virtual prototyping, physical prototyping, or a hybrid form. For example, a subsystem supplier can complement a subsystem with a prototype that assists SoC integrators in the development of application software in parallel with hardware implementation. SoC integrators can then execute their application software together with the subsystem plug-in on a prototype of their host processor while having the prototype of the subsystem hardware execute the production software that comes with the subsystem. Next to software development, prototyping is constructive in support of validation as well as demonstration.

IV. Conclusions

Subsystems pre-integrate hardware and software IP to implement complete system functions and offer high-level hardware and software interfaces for easy SoC integration. This was illustrated with a concrete audio subsystem. Using such subsystems, SoCs can be integrated in a modular fashion with less effort and time-to-market can be accelerated. The use of pre-verified subsystems further helps to reduce risk.

Configurability of subsystems enables reuse across SoCs. Subsystems can provide software plug-ins with a high-level API to support integration into a software stack on a host processor while making core-crossings transparent for the application programmer. We conclude that subsystems can indeed be the next reuse paradigm for efficient SoC integration. This will include the reuse of third party subsystems, in particular for commodity system functions.

References

[1] C.H. van Berkel, "Multi-core for mobile phones", Proceedings DATE'09, pp. 1260-1265, 2009.

[2] Semico Research, "IP subsystems: the next IP market paradigm", October 2010.

[3] Y. Xu, "Employing ARM NEON in embedded system's audio processing", EE Times Asia, January 2010.

[4] Wikipedia, "ARM architecture", http://en.wikipedia.org/wiki/ARM_architecture

[5] www.arm.com

[6] www.synopsys.com

[7] Wikipedia, "I^{2}S", http://en.wikipedia.org/wiki/I%C2%B2S

[8] Wikipedia, "S/PDIF", http://en.wikipedia.org/wiki/Spdif

[9] http://gstreamer.freedesktop.org/

[10] http://www.khronos.org/openmax/

[11] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: why reuse is so hard", IEEE Software, Volume 12, Issue 6, November 1995.

[12] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: why reuse is still so hard", IEEE Software, Volume 26, Issue 4, July-August 2009.