Center for Embedded Computer Systems
University of California, Irvine

Design of a MP3 Decoder using the System-On-Chip Environment (SCE)

Andreas Gerstlauer
Dongwan Shin
Samar Abdi
Pramod Chandraiah
Daniel D. Gajski

Technical Report CECS-07-05
November 2, 2007

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

{gerstl,dongwans,sabdi,pramodc,gajski}@cecs.uci.edu
http://www.cecs.uci.edu

Design of a MP3 Decoder using the System-On-Chip Environment (SCE)

Andreas Gerstlauer
Dongwan Shin
Samar Abdi
Pramod Chandraiah
Daniel D. Gajski

Technical Report CECS-07-05
November 2, 2007

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

{gerstl,dongwans,sabdi,pramodc,gajski}@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

Electronic system-level (ESL) design is touted as a promising solution to sustain productivity in embedded system design in the presence of increasing complexities and decreasing time-to-market. The System-On-Chip Environment (SCE) provides such a SpecC-based ESL design solution. In this report, we demonstrate SCE as applied to the design of a MP3 decoder. Starting from a reference C code, an initial specification model is developed and several different architectural alternatives are explored for implementation on an ARM-based target platform. Using SCE, models for all alternatives are generated and a final, optimal multi-processor system-on-chip (MPSoC) design is selected.

Results of the SCE-based design process show the feasibility and benefits of the approach. Using SCE refinement and exploration tools, all models were generated within minutes. Including the time needed for model simulations, the overall exploration process was completed within an hour. Therefore, the design example demonstrates the capabilities of SCE for rapid, early design space exploration resulting in significant productivity gains.

Contents

List of Figures

List of Tables

Design of a MP3 Decoder using the System-On-Chip Environment (SCE)

A. Gerstlauer, D. Shin, S. Abdi, P. Chandraiah, D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
{gerstl,dongwans,sabdi,pramodc,gajski}@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

Electronic system-level (ESL) design is touted as a promising solution to sustain productivity in embedded system design in the presence of increasing complexities and decreasing time-to-market. The System-On-Chip Environment (SCE) provides such a SpecC-based ESL design solution. In this report, we demonstrate SCE as applied to the design of a MP3 decoder. Starting from a reference C code, an initial specification model is developed and several different architectural alternatives are explored for implementation on an ARM-based target platform. Using SCE, models for all alternatives are generated and a final, optimal multi-processor system-on-chip (MPSoC) design is selected.

Results of the SCE-based design process show the feasibility and benefits of the approach. Using SCE refinement and exploration tools, all models were generated within minutes. Including the time needed for model simulations, the overall exploration process was completed within an hour. Therefore, the design example demonstrates the capabilities of SCE for rapid, early design space exploration resulting in significant productivity gains.

1 Introduction

In the presence of ever-increasing system complexities and time-to-market pressures, the design of embedded systems is facing a growing productivity gap. New methods and tools are needed to sustain the required productivity. Electronic system level (ESL) design has been touted as one of the most promising solutions. ESL approaches aim to close this gap by raising the design process, supported by corresponding design automation tools, to higher levels of abstraction.

The System-On-Chip Environment (SCE) is such a comprehensive ESL design solution for taking a complete embedded system design from initial specification down to its final implementation. SCE supports a wide range of applications and target platforms for design of homogeneous multi-core or heterogeneous multi-processor systems-on-chip (MPSoCs). In SCE, the system is gradually synthesized through a series of interactive exploration and automated refinement steps. Leveraging human insight for crucial design decisions while automating tedious and error-prone tasks like model rewriting enables SCE to deliver the required productivity gains for rapid and early design space exploration. Furthermore, SCE provides an automated path all the way from high-level specification down to hardware/software implementation.

SCE is based on the SpecC system-level design language (SLDL) [5], and it follows a *specify-explore-refine* methodology [8]. The design process starts from a model specifying the design functionality (*specify*). At each following step, the designer first explores the design space (*explore*) and makes the necessary design decisions. SCE then automatically generates a new model at the next lower abstraction level by integrating the decisions into the previous model (*refine*).

An overview of SCE is shown in Figure 1 [1]. The design process starts with a specification model. In the general case, the specification model is an ab-

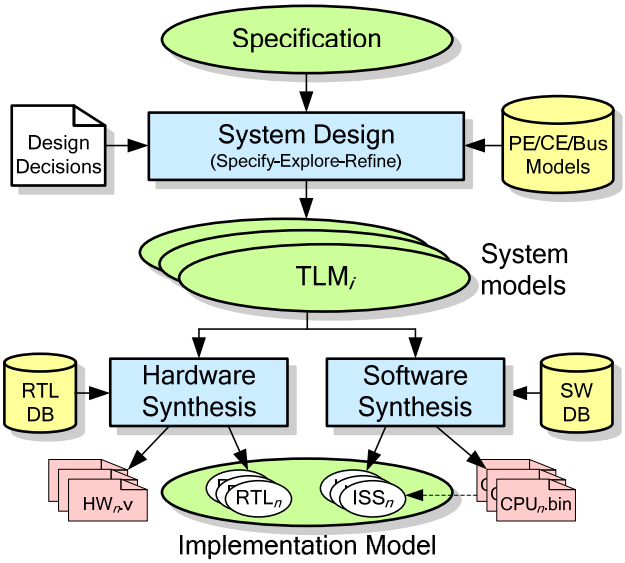


Figure 1: System-On-Chip Environment (SCE).

stract, high-level description of the desired functionality, free of any implementation details [7]. Following a series of system exploration tasks, the specification is then gradually and stepwise refined into transaction-level models (TLMs) of the design at varying levels of abstraction [13, 17, 15, 16]. In each step, the designer enters relevant design decisions through a graphical user interface (GUI) or using SCE's scripting capabilities [4]. Refinement tools then automatically generate a new model implementing and reflecting the user's decisions. In the process, the system is defined, synthesized and assembled using models of available system components taken out of a set of processing element (PE), communication element (CE) and bus databases [6]. As a result, with each exploration and refinement step, a new layer of implementation detail is introduced.

All models in the SCE design flow are represented in SpecC form. As such, models at any stage are executable and can be simulated for validation and feedback about design quality. Intermediate models in the flow allow for early and fast validation of critical design aspects. In general, in an iterative process, designers can vary decisions, generate models and evaluate effects through simulation or analysis until an optimal system design has been reached.

The final pin-accurate model (PAM) of the selected design solution can then be fed into a backend process for further hardware and software synthesis of each individual system component. On the hardware side, high-level synthesis (HLS) of the behavioral, bus-functional description of each hardware component in the PAM into a register-transfer level (RTL) implementation is performed. In addition, SCE supports fully automatic synthesis of software for each programmable processor in the system. Target-specific code is generated, compiled and linked against OS and other libraries taken out of a software database. For each processor, final processor binaries are generated and an instruction-set simulator (ISS) running the target binary is re-integrated into the system model.

As a final result of the SCE design flow, the implementation model at the output of the backend process is a fully cycle- and pin-accurate description of the system design. Furthermore, Verilog/VHDL code and target binaries generated for each hardware and software processor, respectively, provide the data for final logic synthesis, manufacturing or FPGA-based prototyping of the design.

In this report, we demonstrate the System-On-Chip Environment (SCE) as applied to the design of a typical embedded system: an MP3 decoding algorithm as used in cell phones or MP3 players. Starting from the initial C reference code we obtained from [12], we developed the SpecC specification model of the design as a starting point for the design and exploration process (Section 2). Given the specification, we explored several different target architectures for implementation of the decoder on an ARM-based platform (Section 3). Using SCE tools, models of all candidates were generated and evaluated, and an optimal architecture was selected (Section 4). As a result of the design and exploration process, the automatically generated pin-accurate model of the chosen system design is ready for final implementation through further backend hardware and software synthesis.

2 Specification

We started the design process by developing a SpecC specification model of the MP3 decoding algorithm based on an open-source C reference implementation we obtained from the internet [12]. Due to the fact

that SpecC is a complete superset of regular ANSI C, any C code can serve as an initial SpecC model of the application. However, in order to be able to synthesize the code and efficiently explore the design space, the initial C code needs to be converted into a proper SpecC specification [9].

C to SpecC conversion needs to follow a process of stepwise refinement of the code. First, we performed a general cleanup of the C code in order to improve synthesizability at the level of individual expressions. Starting at the top level, we then gradually converted the C functional call hierarchy into a corresponding SpecC hierarchy, introducing and exposing structural and behavioral dependencies, and replacing ambiguous C constructs with their explicit SpecC equivalents.

The resulting SpecC specification model of the MP3 decoder has 14,045 lines of code distributed over 44 behaviors (out of which 29 are leaf behaviors). Conversion from C to SpecC code took approximately 6 man-weeks, out of which 2 man-weeks were spent on initial C code cleanup and the remaining 4 man-weeks on C-to-SpecC hierarchy conversion.

2.1 Reference C Code

The original MAD C code we obtained from [12] is "a new implementation of the ISO/IEC standards that is unencumbered by the errors of other implementations" and "not a derivation of the ISO reference source or any other code." The authors claim that considerable effort has been expended to ensure a correct implementation, even in cases where the standards are ambiguous or misleading.

We chose the MAD library as the basis for our MP3 decoder implementation because it is based on 100% fixed-pointer (integer) computation, allowing it to be implemented even on target processors without a floating-point unit. All calculations in the decoder are performed with a 32-bit fixed-point integer representation. The MAD implementation we started from internally supports 24-bit PCM output for increased precision and high-quality output. The SpecC model, however, only produces 16-bit stereo PCM output, using simple rounding, clipping, and scaling of MAD's high-resolution samples down to 16 bits. As a basic implementation, it does not employ any dithering or noise shaping, which could increase the audible dynamic range based on the extra resolution available internally.

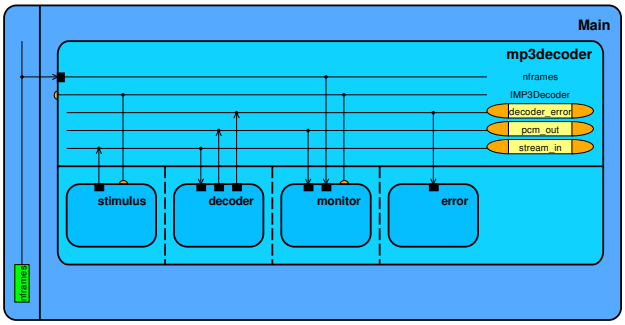


Figure 2: Top-level of MP3 SpecC specification model.

The initial MAD implementation needed considerable cleanup before converting it into a specification model in SpecC. First of all, MAD was a complete MP3 player and included features such as ID3 tag processing. We were only interested in the core MP3 decoder functionality. As a first step, we derived a light weight implementation of the core MP3 decoder with a simple user interface by eliminating unnecessary files and functions. The platform-specific optimizations included by MAD were also removed. Code that depended on advanced C library functions were eliminated if they were determined not to affect the decoding functionality. Further, the MAD implementation used function pointers for some call-back functions. These function pointers were replaced with the calls to the actual functions. Dynamic memory allocations were analyzed and were replaced with safe static allocations. This initial cleanup phase took approximately 2 man-weeks.

2.2 SpecC Model

Conversion of the C code to a SpecC hierarchy starts at the top level of the C functional call hierarchy, i.e. at the `main()` method. In the first step, the `main()` method is converted into an equivalent SpecC *Main* behavior. In the process, the testbench part of the application has to be separated out from the actual parts to be designed.

Figure 2 shows the results of this process for the MP3 decoder SpecC model. At the top-level, the

MP3 decoder *Main* behavior simply executes the *mp3decoder* application, supplying the parsed command line arguments like input and output file names via interface method calls and ports. Internally, the top-level *mp3decoder* executes the actual *Decoder* design next to testbench behaviors for file I/O including supplying input stimuli, monitoring resulting outputs, and checking for fatal error exit conditions.

As is typical for a specification testbench setup, the testbench part (stimuli and monitor behaviors) run concurrently to the actual design. *Stimulus* and *Monitor* behaviors are supplied with the names of input and output files to read from and write to through ports connected to the overall *Main* behavior. Internally, the design communicates with the testbench through abstract channels for incoming MP3 bytes (*stream_in*), outgoing PCM samples (*pcm_out*) and asynchronous error conditions (*decoder_error*). All three channels are FIFO queues for buffering of frame data and decoupling of threads in order to improve performance.

The actual *Decoder* application to be designed (Figure 3) is at its highest level a finite state machine, captured in a SpecC `fsm` composition. After initialization and setup of the decoder (*init* and *mute* states), the decoder successively starts reading bytes from the input MP3 stream channel (*stream_in*) until a complete frame has been received. Since MP3 frames can be of variable length and since a frame's main data can include data from past and/or future frames (negative and positive main data offset aka MP3 bit reservoir), the frame header needs to be decoded in order to determine how many bytes need to be read from the input stream. Therefore, *input* and *decode_header* states run in a loop until a complete frame has been internally buffered (in the local *stream* array). Once the *header* has been completely decoded and a complete frame of data is available, decoding continues with processing of the frame body (*decoder_frame*). The result of frame decoding is the set of decoded subband samples for the given frame (*frame_sbsample*). Finally, a last *synth* state performs full frequency PCM synthesis to produce the final PCM samples sent out over the *pcm_out* channel.

After a frame has been successfully decoded and a final error check has been performed (*recover*), the *Decoder* loops back to the setup state (*mute*) in order

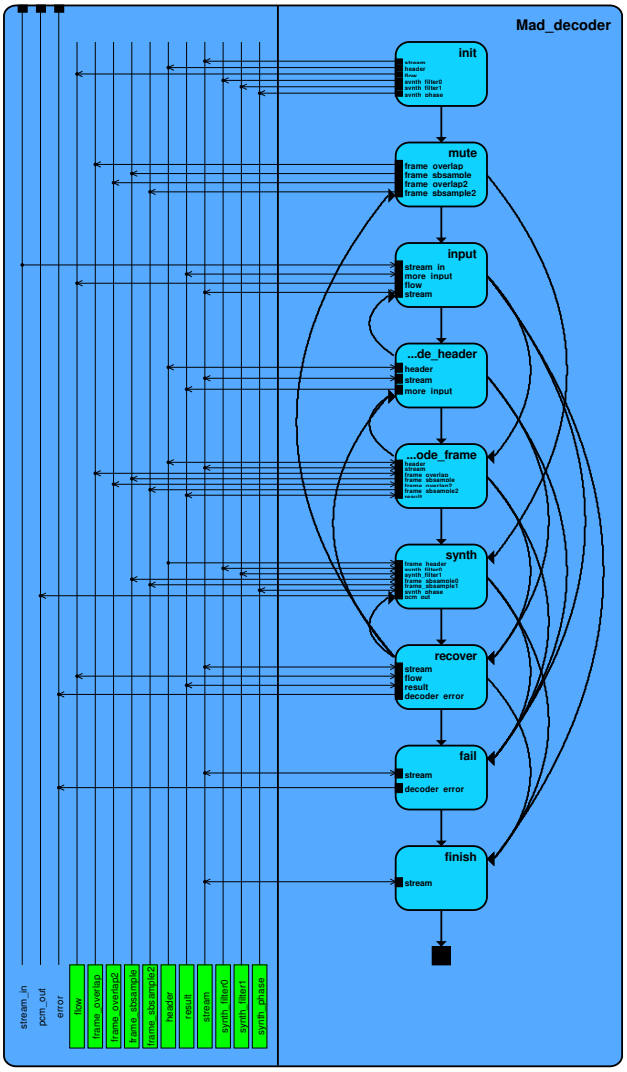


Figure 3: Behavioral and structural hierarchy of MP3 decoding.

the begin processing of the next frame. In general, errors can occur at any stage of the decoding process. Errors are subdivided into recoverable and fatal errors. In each state, decoding errors, e.g. due to invalid input data, are checked. If a recoverable error is detected, the state machine branches directly to the *recover* state, which will try to restore the internal state variables of the decoder to a sane status before jumping ahead to the decoding of the next frame. Since the start of a frame in the input stream is not known, the *decode_header* state (in a loop with the *input* state) will try to re-synchronize decoding in such situations

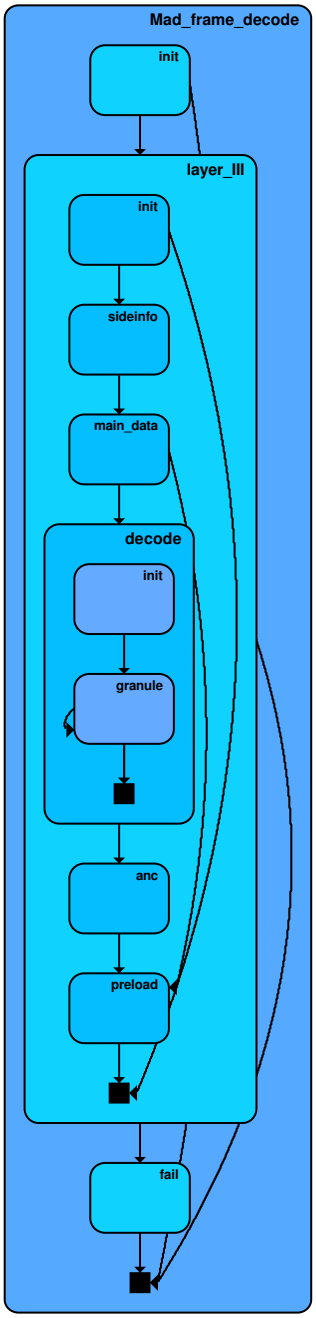


Figure 4: Behavior hierarchy of MP3 frame decoding.

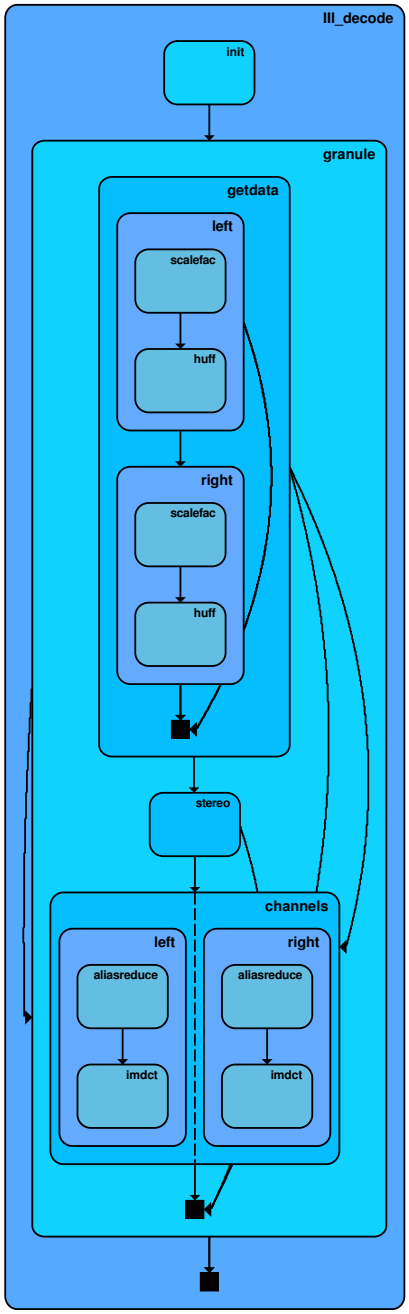


Figure 5: Behavior hierarchy of granule decoding in an MP3 frame.

by scanning the input stream until the next synchronization word marking the start of a frame is received. In all cases, if recovery or synchronization fails, or if a fatal error occurs at any stage of the decoding process, the *Decoder* will branch to the *fail* state before exiting the decoding completely (*finish*). Also, the decoder will *finish* if the end of the stream has been reached, i.e. no more *input* is available.

2.2.1 Frame Decoding

The internal decomposition of the frame decoding state is shown in more detail in Figure 4. The *Frame_decode* behavior is sequentially composed out of a multi-level hierarchy of subbehaviors. At the topmost level, frame decoding consists of a state machine that performs the actual MPEG layer III decoding enclosed by initialization and error handling states. The *Layer_III* behavior is then further decomposed into a state machine for the actual decoding and signal processing chain. Layer III processing consists of states for initialization (*init*), decoding of frame side information (*sideinfo*), preprocessing and collection of frame main data (*main_data*), core layer III decoding (*decode*), designation of ancillary bits (*anc*), and preloading of next frame's data (*preload*). Furthermore, the core decoding is further subdivided into a state machine that, after some initialization, sequentially loops over the two granules that form the core of each MP3 frame.

Figure 5 shows this core layer III decoding with granule processing (*granule*) further expanded. Decoding of each of the two granules is subdivided into three major states: data demultiplexing and decompression (*getdata*), joint stereo decoding (*stereo*) and stereo channel decoding (*channels*). Note that the *stereo* step is skipped if the stream does not use joint stereo encoding. Both *getdata* and *channels* decoding internally each contain two instances of basic single channel processing behaviors, one instance each per *left* and *right* channel. Data demultiplexing and decompression consists of scalefactor (*scalefac*) and huffman (*huff*) decoding for each channel. On the other hand, decoding of each stereo channel consists of anti-aliasing (alias reduction, *aliasreduce*) and an IMDCT (*imdct*) for conversion from the frequency into the time domain.

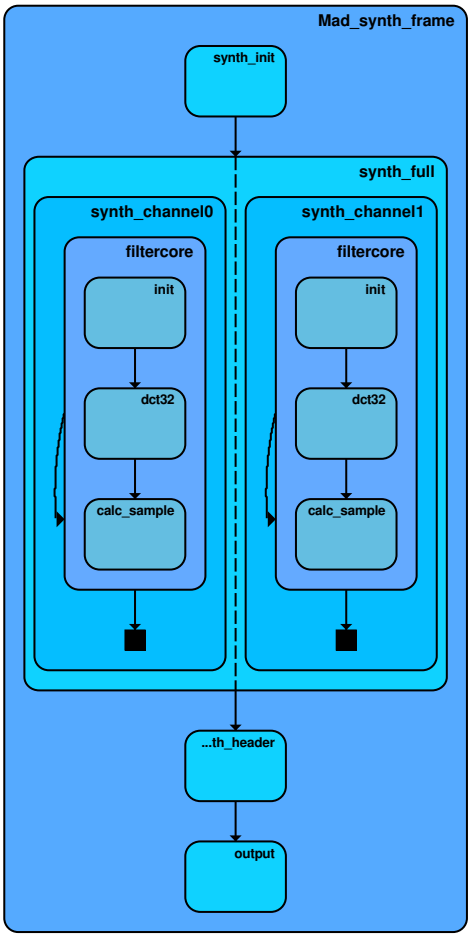


Figure 6: Behavior hierarchy of MP3 PCM synthesis.

For *channels* decoding, both channels can be decoded in parallel. On the other hand, since internal state is kept during huffman decoding, successive invocations of *huff* are not independent of each other. Therefore, neither the two channels in the *getdata* block nor the two granules themselves can be decoded concurrently, i.e. due to the dependencies across invocations, the correct sequential order of huffman decoding calls needs to be maintained.

Finally, note that on each level of frame decoding, error handling is performed asynchronously to the regular decoding chain. If the stream is in error condition or if errors are detected during input data processing (*main_data* or *granule*'s *getdata*), the global error conditions is set and decoding is aborted, i.e. the control flow transitions to the exit state through each level of the hierarchy.

2.2.2 PCM Synthesis

The last stage of the MP3 decoding process is the final synthesis of the PCM samples, shown in more detail in Figure 6. At the top level, the *Synth_frame* behavior sequentially performs the actual full-frequency PCM synthesis (*synth_full*), preceded and followed by small blocks for variable initialization (*synth_init*) and header post-processing (*synth_header*), respectively.

The *synth_full* behavior is then further composed out of two instances of a *synth_channel* behavior running in parallel. Each of the two instances is independently responsible for synthesizing one of the two stereo output channels from the given input array of subband samples. Internally, each channel behavior executes the respective synthesis filter core (*filtercore*) repeatedly in a loop, once for each group of 32 output samples to be produced per channel. Synthesizing each block of 32 samples then consists of an initialization step (*init*), a discrete cosine transformation (*dct32*), and rounding, clipping and quantization of samples (*calc_sample*).

Finally, following the core PCM synthesis, the final *output* behavior then takes the internally buffered samples and sends them to the testbench via the output FIFO queue *pcm_out*.

3 Design Space Exploration

Given the specification model, we investigated a realization of the MP3 decoder design on an ARM-based target platform [2]. In the process, we explored several different architectural alternatives for implementation of the decoder across software and hardware domains. The main objective of design space exploration was the optimization of the overall MP3 frame decoding delay. Based on the sequential nature of the MP3 specification at its upper layers, optimization was primarily focused on hardware acceleration of critical blocks, with exploitation of available parallelism as a secondary goal.

Using SCE's profiling and estimation capabilities [3], we analyzed the computational complexity of the MP3 decoding algorithm. Combining dynamic profiling of the input specification model simulation with a static analysis and matching of specification and

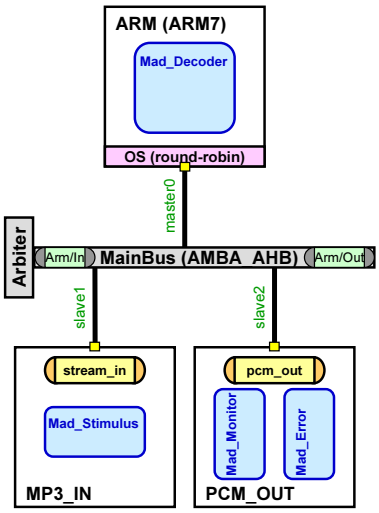
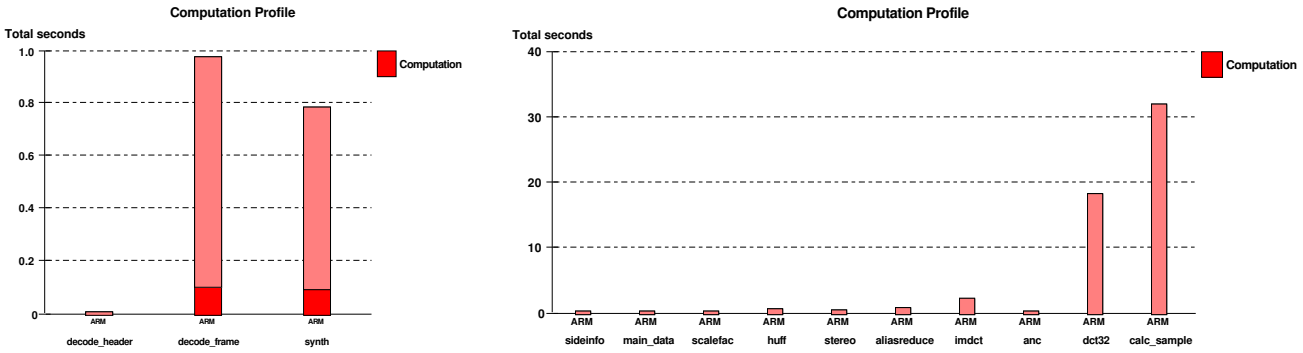


Figure 8: MP3 platform with pure software implementation (SWPE).

ARM processor characteristics, the SCE profiler generates estimates about the software execution times of each MP3 block on the given ARM processor. Figure 7 shows the estimated total delays per block obtained for a decoding of eight MP3 frames. As shown in Figure 7(a), overall MP3 decoding delays are almost evenly distributed between frame decoding and PCM synthesis. Looking at individual leaf behaviors of the frame decoding and PCM synthesis blocks (Figure 7(b), major contributors in each block are the *Imdct* and *Dct* behaviors, respectively. Furthermore, the regular and inverse modified discrete cosine transforms (DCT and IMDCT) are both widely-used, general digital filter algorithms, i.e. advanced hardware implementations are readily available. As such, they make good candidates for hardware acceleration.

3.1 Pure Software Implementation

We started the design and exploration process by investigating a pure software solution of the MP3 decoding algorithm running on an ARM7TDMI target processor. As shown in Figure 8, in this most basic target architecture, the ARM processor runs the main *Decoder* behavior on top of a real-time operating system. In our case, we chose μC/OS-II [18] using a round-robin scheduling strategy as target operating system. As part of the scheduling exploration step in



(a) top-level

(b) leaf behaviors

Figure 7: Computational complexity of MP3 decoder blocks.

the SCE design process, all parallel behavior compositions for left and right channel processing inside the *granule* decoding and PCM *synth* blocks were refined into dynamically spawned tasks running concurrently on top of the OS. For validation of and feedback about dynamic scheduling effects during simulation, SCE automatically inserts an abstract, high-level model of the chosen operating system into the generated TLM and PAM design models [11].

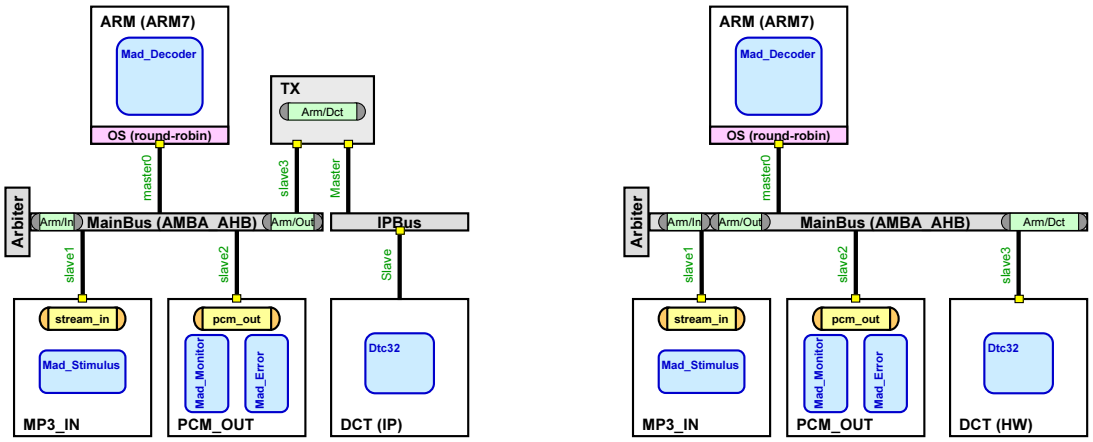
In addition to the actual decoding algorithm running on the ARM, the processor is assisted by two hardware I/O units for MP3 stream input and PCM speech sample output processing and buffering. As such, the *Stimulus*, *Monitor* and *Error* behaviors of the top-level MP3 design specification are mapped to *MP3_IN* and *PCM_OUT* I/O units, respectively. Furthermore, note that the two FIFO queues for stream input and PCM output between the decoder and the monitor and stimulus behaviors have each been mapped into the corresponding hardware unit for implementation. Using SCE, the queues will therefore be implemented as local send and receive FIFOs inside each of the hardware I/O processors.

The ARM processor and the I/O blocks communicate over a single instance of an AMBA AHB local processor bus. The ARM processor is a master on its bus and the two I/O units are synthesized to connect as AHB slaves. As such, all communication between the ARM processor and the I/O units will be routed over the AHB bus. Specifically, the decoder running on the ARM processor will read input MP3 stream data from and write output PCM samples to the hardware FIFOs in the *MP3_IN* and *PCM_OUT* I/O blocks, respectively. All communication between the ARM processor and the I/O queues is implemented by mapping FIFO registers and link channels into the AHB address space using dedicated bus addresses and processor interrupt lines.

3.2 DCT Hardware Acceleration

In the first step of the exploration and optimization process, we chose the critical *Dct32* component (see Figure 7(b)) as the candidate for hardware-assisted acceleration in a co-processor fashion. The *Dct32* was mapped into a separate, stand-alone hardware PE that acts as a slave to the main decoding algorithm running on the master ARM processor. As a result, when reaching the corresponding stage in the decoding process, the ARM will send the input data to the *DCT* hardware component for processing. The ARM software will then wait for the results coming back from the DCT before continuing with the decoding process. While waiting for the DCT, the operating system on the ARM can swap in and switch over to any other ready task, thus exploiting available parallelism on the software side, if any.

The resulting system computation and communication architecture is shown in Figure 9. In this first step, we allocated only a single *DCT* co-processor. The single *DCT* hardware PE is shared between both left and right channel PCM synthesis processes on the ARM (running two independent, concurrent loops for processing of samples each, see Figure 6). All in the hope to exploit the parallelism (across loop itera-



(a) IP-based DCT hardware.

(b) Synthesized DCT hardware.

Figure 9: MP3 platform with DCT hardware accelerator (HWSW1).

tions) between *calc_sample* processing of one channel on the ARM while waiting for the *dct32* of the other channel being processed in hardware (and vice versa).

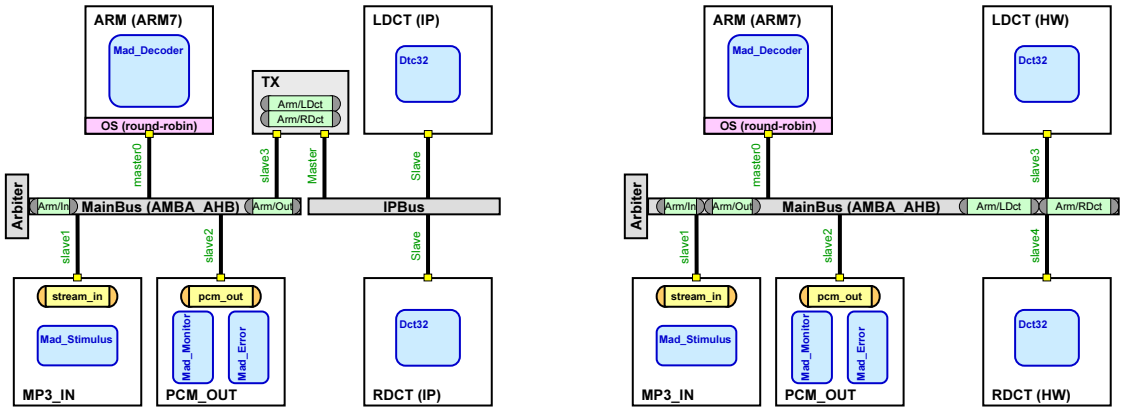
In the process of DCT acceleration, we explored two different architectural variants: reuse of a predesigned *DCT* IP component (Figure 9(a)), and synthesis of a fully-custom *DCT* hardware unit from scratch (Figure 9(b)). In the latter case, the custom hardware unit can be synthesized with SCE to implement any bus interface. Therefore, it can be directly connected to and implement the AHB slave protocol. In this case, all communication for sending input blocks from the ARM to the DCT and for receiving transformed results back from the DCT to the ARM will go over the main AHB bus. Following the co-processor principle, the DCT is a slave on the main system bus and the ARM controls all transfers as the single master on its bus. To send events for status updates to the ARM processor, the DCT hardware generates interrupts and is connected to selected processor interrupt input lines.

In the IP case, the *DCT* IP component is directly connected to and comes with its own local, dedicated *IPBus*. A transducer is then added to the communication architecture, translating between the two protocols and connecting the IP component to the main AHB bus. As such, all communication between the ARM processor and the DCT IP has to go over the transducer and the two busses. Again, the ARM is the single master on its bus. The transducer is a slave on the system bus and a master for the IP, relaying all ARM request for sending and receiving of data to the IP. Again interrupts are used for event notification from the DCT to the ARM, relayed by the transducer.

3.3 Parallelized DCT Hardware Acceleration

The next logical step in hardware acceleration is to duplicate the *DCT* unit in order to provide dedicated co-processor instances for each of the two channels. Theoretically, including two independent *LDCT* and *RDCT* hardware PEs enables further exploitation of available parallelism by allowing to run the two DCT instances in the left and right channel concurrently at the same time.

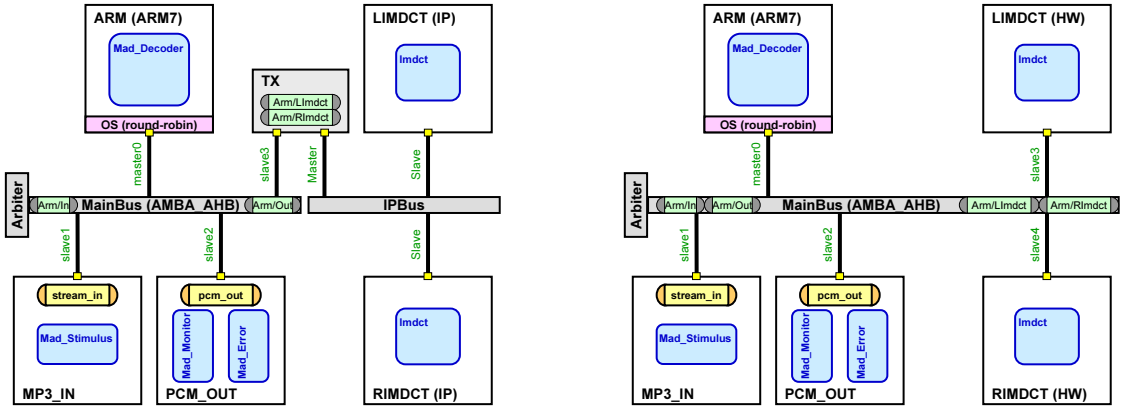
Figure 10 shows the corresponding architectures with two DCT units, one for each channel. Again, we implemented both an IP-based architecture with transducer (Figure 10(a) and an architecture with synthesizable DCT hardware PEs directly connected to the AHB bus (Figure 10(b)). In the former case, it was assumed that both DCT IPs can be connected to a shared instance of a single *IPBus*. Without loss of generality, an architecture with two separate, dedicated busses for each IP would be a simple, straightforward extension that is not shown here.



(a) IP-based DCT hardware.

(b) Synthesized DCT hardware.

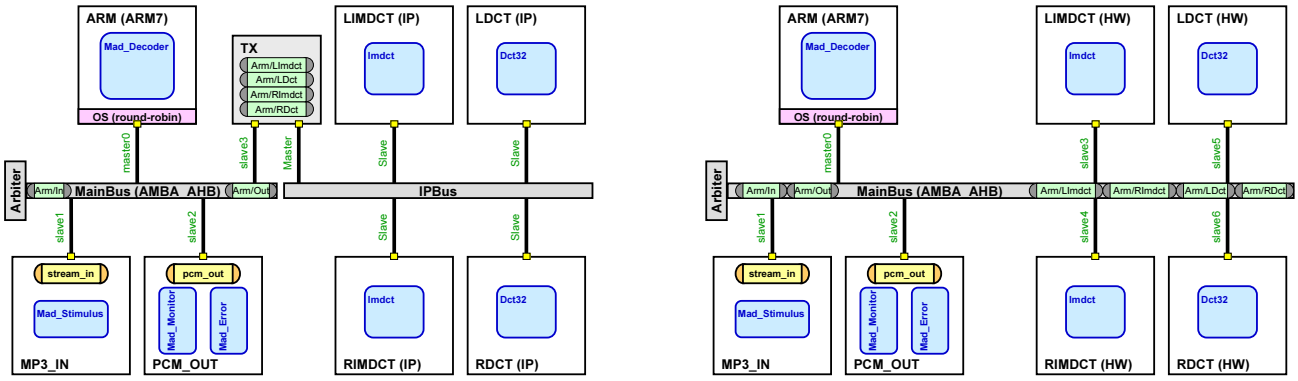
Figure 10: MP3 platform with concurrent DCT hardware accelerators (HWSW2).



(a) IP-based DCT hardware.

(b) Synthesized DCT hardware.

Figure 11: MP3 platform with concurrent IMDCT hardware accelerators (HWSW3).



(a) IP-based DCT hardware.

(b) Synthesized DCT hardware.

Figure 12: MP3 platform with DCT and IMDCT hardware accelerators (HWSW4).

3.4 Parallelized IMDCT Hardware Acceleration

In addition to acceleration of the PCM synthesis stage, we investigated options for decreasing the delay in the frame decoding stage of the MP3 decoding algorithm. As shown in Figure 7(a), both stages are similar in their workload and contribution to overall frame delays. Furthermore, from Figure 7(b), it can be concluded that the *Imdct* block is the most critical leaf behavior in the frame decoding stage, i.e. the primary candidate for hardware acceleration there.

In order to be able to explore effects of frame decoding optimizations independent of and unaffected by PCM synthesis modifications, we first explored acceleration of only the *Imdct* block alone. Similar to the *Dct32* behavior in the synthesis stage, the MP3 specification executes two instances of the *Imdct* concurrently as part of parallel left and right channel decoding (Figure 5). Note, however, that even though concurrent channel decoding is executed in a loop over the two granules that are part of each MP3 frame, parallelism does not extend across loop iterations. As specified, both threads have to be joined before the next loop iteration can be started.

Therefore, we did not further investigate a solution with a single IMDCT hardware unit shared across channels. Instead, we directly implemented an architecture with separate, dedicated *LIMDCT* and *RIMDCT* PEs allowing for maximal parallelism between the left and right channel decoding tasks, respectively.

The result of that exploration step is shown in Figure 11. Following the two variants for the DCT case, we explored both an IP-based (Figure 11(a)) and a custom synthesized (Figure 11(b)) version of the IMDCT hardware units. In all cases, IMDCTs operate as co-processors, i.e. they act as slaves to the single master ARM processor on the main system bus. In the IP case, the IMDCTs are connected to their own *IPBus* and a transducer connects the IP bus as slave to the main bus. Interrupts are used for synchronization and event notification from IMDCT PEs to the ARM processor in each case.

3.5 DCT and IMDCT Hardware Acceleration

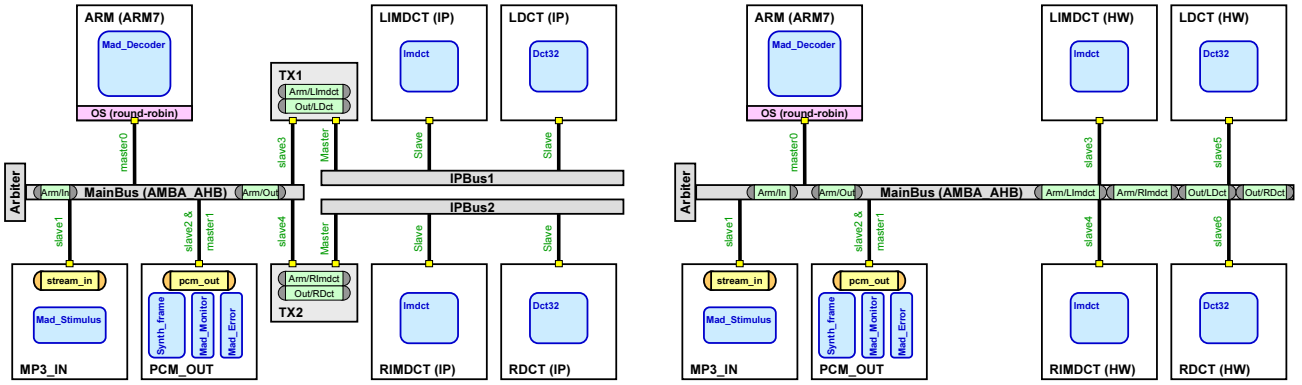
Based on the fully parallelized and hardware accelerated architectures presented in Section 3.3 and Section 3.5, we created a first combined, maximally parallel system architecture which includes both DCT and IMDCT co-processors. To allow exploitation of all potential and available concurrency, the resulting system includes separate, dedicated *LDCT*/*LIMDCT* and *RDCT*/*RIMDCT* hardware PEs for left and right channels, respectively (Figure 12).

Similar to previous cases, we created architectures with both IP-based and synthesized implementations of DCT and IMDCT co-processors (Figure 12(a) and Figure 12(b), respectively). In the IP-based solution, all four IP components are assumed (without loss of generality) to be connected as slaves to a common, shared *IPBus* instances that is connected to the main system bus via a transducer. In the case of synthesizable custom hardware components, all four co-processors are directly connected as and synthesized to become slaves on the AHB bus. In all cases, co-processors are slaves listening to and generating interrupts for a sole master ARM processor.

3.6 Pipelined DCT and IMDCT Hardware Acceleration

Even in the most parallel system architecture (Section 3.5), the available concurrency within each MP3 frame is limited by the sequential nature and the inherent dependencies of the MP3 decoding algorithm. We can, however, increase performance further by pipelining the decoding algorithm in order to expose and exploit additional parallelism that is available across successive MP3 frames. This can be achieved by splitting the decoder into two parts distributed across different processors such that the software on the ARM processor can start processing the next frame while the hardware is finishing the last synthesis and PCM output stages of the current frame.

In the MP3 decoder this is possible because the only dependencies are between stages in the same frame (data passing from one stage to the next) and inside the same stage across frames (huffman and synthesis filter state kept across iterations). Specifically,



(a) IP-based DCT hardware.

(b) Synthesized DCT hardware.

Figure 13: MP3 platform with pipelined DCT and IMDCT hardware accelerators (HWSW).

no dependencies exist between the PCM synthesis stage and the header or frame decoding stages of the next frame. Therefore, in order to implement pipelining of those two stages, the complete PCM synthesis and output block has to be mapped into hardware.

The resulting final, pipelined and parallelized MP3 decoder system is shown in Figure 13. As described in Section 3.1, the *Output* and *Error* stages were already previously mapped into a separate *PCM_OUT* hardware. In order to enable pipelining, we also map the complete *synth_frame* block into the same *PCM_OUT* PE. As such, *synth_frame* and *Output* behaviors communicate PCM samples through the HW-local *pcm_out* queue internally without involving the ARM processor. This allows the PCM synthesis and output stages to run completely independent of the software on the ARM.

The frame decoding software on the *ARM* and the PCM synthesis hardware in the *PCM_OUT* PE are each assisted by respective co-processors for IMDCT and DCT acceleration. Again, co-processors are duplicated to include two dedicated PEs each for left and right channel processing. The ARM processor communicates with the *LIMDCT* and *RIMDCT* components whereas the *PCM_OUT* PE exchanges data with the two DCTs. Since all four co-processors are acting as slaves, the *PCM_OUT* has to become both a slave (for communication with the ARM) and a secondary master (for communication with *LDCT* and *RDCT*) on the main system bus. As such, the AHB bus implementation needs to include a mandatory bus arbiter component.

As in all previous explorations, we implemented two architectures using either IP components (Figure 13(a)) or synthesizing custom hardware (Figure 13(b)) for each co-processor. In the latter case, all four co-processors are synthesized to become AHB bus slaves directly connected to the main system bus. In the former case, IP components require separate instances of their own, proprietary IP bus protocol. Transducers then connect the IP busses to the AHB system bus, translating between the two.

In contrast to all previous architectures with a single ARM master component only, the two AHB bus masters (*ARM* and *PCM_OUT*) can access co-processors concurrently. Therefore, to avoid potential contention of concurrent accesses by competing masters on a single, slow IP bus, we allocated two separate IP busses connected via two transducers. Co-processors are evenly distributed and connected to the two IP busses based on a separation of left and right channels. The two separated busses allow the *ARM* and *PCM_OUT* masters to concurrently access different channel co-processors each, removing the potential bottleneck of a single, shared IP bus[1].

[1]Note that due to the speed difference between the AHB and IP busses, transactions are buffered in the transducers based on a store-and-forward principle. Therefore, the AHB bus is not a bottleneck. Even though they are serialized by arbitration, AHB masters can fill or empty the buffers in the transducers faster than and while transducers perform slow transactions on the IP busses.

4 Refinement Results

Going through the different exploration and refinement steps of the SCE design and tool flow, we realized the design implementations for all explored system architectures as described in Section 3. Using SCE's automatic model generation and refinement capabilities, transaction-level and pin-accurate models (TLM and PAM) at varying levels of abstraction were automatically generated for each of the design alternatives [10]. Using SCE tools, models for all target implementations were generated within minutes. Furthermore, including time needed for validation and simulation of models, the complete design space exploration process was completed in less than an hour.

In all cases, we brought down the implementation to a final pin-accurate model ready for further hardware and software synthesis. For final sign-off, all models were executed for validation through simulation. Model simulations were performed on a 2.8 GHz Intel Pentium 4 workstation running Linux. Validation of models was based on a testbench that exercises the MP3 design by decoding 10 frames of a stereo MP3 test stream with 44.1 kHz sampling frequency and a (constant) bitrate of 96 kbit/s. Note that since a MP3 frame consists of 1152 PCM samples, each frame corresponds to 26.12 ms and the total decoded stream length for this setup is 0.2612 s of audio.

Results for the pin-accurate models of all explored system architectures and for the initial specification model are summarized in Table 1. For each model, the table shows model statistics such as lines of code (LOC) and number of behaviors (overall and leaf) and channels. In addition, the time needed to simulate the model, the simulated MP3 frame decoding delay and the total runtime of the refinemenent tools for generation of the model are listed.

As is always the case for a purely functional model, the initial specification executes in zero time, i.e. with a frame delay of zero. Furthermore, specification simulation is very fast, running the algorithm natively with no extra overhead on the simulation host. All subsequent pin-accurate models, on the other hand, have significantly higher simulation times due to the extra implementation detail and resulting simulation overhead in those models. Note that a large part of the overhead at the pin-accurate level can be attributed to the simulation of events on bus wires. Accurate bus-functional models are necessary for further hardware synthesis. On the other hand, for simulation and validation, those details can be abstracted away. Applying transaction-level modeling (TLM) techniques, speedups of several orders of magnitude with no loss of accuracy can be achieved [14]. However, since we focus on synthesis in this report, we do not include any TLM results here.

In general, model complexities as measured by model statistics (i.e. static code size and number of objects) and simulation times are correlated to the complexity of the target architecture (number of components and busses). On the other hand, simulation times also depend on the amount of dynamically simulated content and functionality, which are directly related to the actual frame decoding delay. Note, however, as discussed above, compared to the computation content, slow simulation of communication functionality at the pin-accurate level will have a proportionally higher impact on simulation speeds than its actual contribution to overall simulated frame delays.

As can be seen from the results, a pure software solution *SWPE* meets the frame deadline (frame decoding constraint) of 26.12 ms, but not comfortably. However, since there is not a big enough margin to compensate for any variations in delays when taking the high-level estimates down to their final implementation, the software implementation is not considered to be feasible. Looking at various levels of hardware acceleration, we can conclude that in all cases IP-based solutions have higher frame delays than their synthesized counterparts. This is due to the extra overhead necessary for translation to/from and communication over the slower IP bus protocol. Hence, there is a clear trade-off between reuse of predesigned IP components (i.e. cost reduction) and delays (i.e. speed).

Results for architecture *HWSW1* and *HWSW2* show that frame delays actually increase when adding DCT hardware acceleration. This can be attributed to the fact that the extra overhead for communication between the ARM and the DCT units outweigh the benefits of speeding up the DCT algorithm in a hardware implementation. The DCT in the PCM synthesis

Model		Statistics				Simulation time	Frame delay	Generation time
		LOC	Behaviors	Leafs	Channels			
Specification		14,045	44	29	2	0.01 s	0.00 ms	—
SWPE		22,085	96	58	29	5.67 s	25.92 ms	4.01 s
HWSW1	IP	24,922	119	72	81	254.9 s	38.67 ms	5.40 s
	HW	23,766	114	68	57	157.6 s	26.34 ms	5.17 s
HWSW2	IP	24,800	119	72	81	255.4 s	38.67 ms	5.93 s
	HW	23,710	114	68	57	158.6 s	26.34 ms	5.60 s
HWSW3	IP	25,140	123	73	85	104.7 s	25.76 ms	7.35 s
	HW	24,042	118	69	61	65.9 s	20.29 ms	7.10 s
HWSW4	IP	27,404	143	85	129	377.9 s	38.51 ms	10.85 s
	HW	25,697	136	79	89	221.1 s	20.71 ms	10.24 s
HWSW	IP	28,528	148	87	150	308.5 s	17.71 ms	12.46 s
	HW	26,847	142	83	110	225.2 s	12.46 ms	11.92 s

Table 1: Exploration and refinement results.

stage is executed in a loop, once for every group of 32 samples. For the 10 frames decoded through the testbench, the *Dct32* behavior is invoked 575 times. On each invocation, a DCT co-processor requires to receive and send about 2 kB of data. Therefore, external DCT processing incurs significant traffic on the bus(es) in each frame. Furthermore, note that a parallel implementation with duplicated DCTs does not bring any benefits as the frame delay is not affected. This shows that delays in the PCM synthesis stage are limited by the software running on the ARM, i.e. any additional speedup of the DCT will not further reduce the overall delay.

Comparing the IMDCT-based architecture *HWSW3* with DCT acceleration, results are different. Hardware acceleration of the *Imdct* block reduces frame delay compared to the software solution. In the IMDCT case, communication overhead does not become a bottleneck and especially the custom synthesized IMDCT PEs markedly improve delays (for IP-based IMDCTs, on the other hand, the additional IP communication overhead and acceleration gains even out).

In the case of architecture *HWSW4*, when combining IMDCT and DCT acceleration results are mixed. Again, adding DCT acceleration increases delays. This effect is generally more pronounced for IP-based DCT implementations due to the extra overhead for IP protocol translation and communication in this case. Therefore, an architecture with custom synthesized co-processors can maintain delay gains whereas the deadline is violated in the IP-based variant.

Finally, only the fully pipelined and parallelized architecture *HWSW* can achieve the required frame delays with a big enough margin. Pipelining of frame decoding on the ARM and PCM synthesis in hardware dramatically reduces overall delays. The parallelism available across frame iterations provides by far the biggest potential for speed gains (at the expense of significantly higher hardware costs). Therefore, architecture *HWSW*, either in its IP-based or synthesized form, was chosen as the final system design for an ARM-based implementation of the MP3 decoder.

5 Summary and Conclusions

In this report, we presented the application of the System-On-Chip Environment (SCE) tool flow to the design of a MP3 decoder system on an ARM-based target platform. The design process starts with a specification model of the MP3 algorithm described in the SpecC system-level design language (SLDL). Using SCE exploration and refinement tools, six different base architectures for implementation of the MP3 design with varying levels of either IP-based or synthesized hardware acceleration were investigated. For all

design alternatives, transaction-level and pin-accurate models were automatically generated. All models were validated and evaluated through simulation. An optimal architecture with pipelined and parallelized hardware acceleration of DCT and IMDCT blocks was obtained as the final system design and optimized MP3 implementation. As a result of the exploration process, the final pin-accurate model of the selected architecture serves as the input to the backend process for further hardware and software synthesis.

Results show the feasibility of the approach and prove the tremendous benefit of a SCE-based electronic system-level (ESL) design solution. Varying models for all design alternatives were automatically generated using SCE refinement tools. As a result, the exploration process was completed and an optimal architecture was selected in less than 1 hour, including time required for model validation and simulation. In summary, significant productivity gains with design times that are several orders of magnitude shorter when compared to traditional manual modeling and design approaches have been achieved.

References

[1] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel Gajski. System-on-chip environment (SCE version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[2] AMBA Home Page. www.arm.com/products/solutions/AMBAHomePage.html.

[3] Lucai Cai, Andreas Gerstlauer, and Daniel D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *Proceedings of the Design Automation Conference (DAC)*, San Diego, CA, June 2004.

[4] Lukai Cai, Andreas Gerstlauer, Samar Abdi, Junyu Peng, Dongwan Shin, Haobo Yu, Rainer Dömer, and Daniel Gajski. System-on-chip environment (SCE version 2.2.0 beta): Manual. Technical Report CECS-TR-03-45, Center for Embedded Computer Systems, University of California, Irvine, December 2003.

[5] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[6] Andreas Gerstlauer, Lukai Cai, Dongwan Shin, Haobo Yu, Junyu Peng, and Rainer Dömer. *SCE Database Reference Manual, Version 2.2.0 beta*. Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[7] Andreas Gerstlauer and Rainer Dömer. *SCE Specification Model Reference Manual, Version 2.2.0 beta*. Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[8] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[9] Andreas Gerstlauer, Kiran Ramineni, Rainer Dömer, and Daniel D. Gajski. System-on-chip specification style guide. Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[10] Andreas Gerstlauer, Donwan Shin, Junyu Peng, Rainer Dömer, and Daniel D. Gajski. Automatic Layer-Based Generation of System-On-Chip Bus Communication Models. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 26(9):1676–1687, September 2007.

[11] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS modeling for system level design. In Ahmed A. Jerraya, Sungjoo Yoo, Norbert Wehn, and Diedrik Verkest, editors, *Embedded Software for SoC*. Kluwer Academic Publishers, 2003.

[12] Underbit Technologies Inc. MAD: MPEG audio decoder. http://www.underbit.com/products/mad.

[13] Junyu Peng, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-Chip Architecture Modeling Style Guide. Technical Report CECS-TR-04-22, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[14] Gunar Schirner and Rainer Dömer. Result Oriented Modeling – A Novel Technique for Fast and Accurate TLM. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 26(9):1688–1699, September 2007.

[15] Dongwan Shin, Lukai Cai, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-Chip Transaction-Level Modeling Style Guide. Technical Report CECS-TR-04-24, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[16] Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-Chip Communication Modeling Style Guide. Technical Report CECS-TR-04-25, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[17] Dongwan Shin, Junyu Peng, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-Chip Network Modeling Style Guide. Technical Report CECS-TR-04-23, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[18] uCos-II. http://www.ucos-ii.com.