Technical Report: Communication SW Generation from TL to PCA Level for MPSoC

Ines Viskic, Samar Abdi and Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine, CA 92617
{iviskic, sabdi, gajski}@ics.uci.edu

July 6, 2007

Abstract

This paper describes the automatic generation of MPSoC communication SW from transaction level (TL) to a pin and cycle-accurate (PCA) level. At the transaction level, the communication in MPSoC is abstracted with send/receive and read/write calls to channels modeling system busses. Since the view of the MPSoC platform is abstracted with the goal of fast simulation and performance estimation, the TLM cannot be compiled and downloaded to the FPGA board. Our communication synthesis tool automatically transforms abstracted TLM communication functions into platform specific IF and drivers in the output PCAM. The output model can then be fed into platform synthesis tools and compilers for automatic download to the board. The results of our experiments demonstrate the effectiveness of our tool. The automatic IF and drivers generation for PCAM yields significant productivity gain over manual design, while maintaining comparable performance (as measured by the communication delay) and code size.

1 Introduction

The increasing performance requirements and application complexity require the advancement of system design into multiprocessor system on chips (MPSoC). MPSoC systems execute faster than traditional single core SoC because of their high computing power and large-scale parallelism. MPSoC usually consist of multiple application-specific, heterogeneous processors (CPUs), multiple units of digital signal processing hardware (DSP units), memory units and controllers, high-speed on-chip communication interfaces and sophisticated communication protocols.

Designing MPSoC with traditional methodologies (top-to-bottom, bottom-up) is inefficient due to systems size and complexity, making it difficult to meet the stringent time to market constrains. Platform based HW/SW co-design is widely seen as a solution to simplifying the MPSoC design process. It consists of separating the application code into multiple concurrent processes, mapping each into predefined SW/HW components in the MPSoC platform and making them communicate correctly. However, manually designing MPSoC communication is becoming an extremely challenging part of platform based HW/SW co-design, being both error prone and time consuming.

This paper describes the automatic generation of platform dependent, application specific communication SW in MPSoC designs from TLM function calls. The communication software synthesis is a part of the integral Synthesis Tool for MPSoCs.

The rest of the paper is organized as follows: Section 2. presents the related work and Section 3 explains the general principles of automatic system level synthesis. Section **??** provides focus on the synthesis of communication SW: its inputs, outputs and synthesis procedure(s). Experimentation in Section 5. demonstrates the effectiveness of automatic synthesis. We conclude the paper with the summary of contributions in Section 6.

2 Related Work

Several research groups target multiprocessor architectures and work to develop an efficient MPSoC design methodology. Jerraya et al. [1] have developed a generic architecture model of MPSoC (GAM-MPSoC). This model is characterized with modularity, flexibility and scalability. Using GAM-MPSoC and binaries for each processor as inputs, the authors are able to synthesize component wrappers and communication channels to generate a detailed, pin-accurate micro-architecture of the system. However, the process is not fully automatic, since the allocation tables needed for SW to HW mapping have to be written manually. This slows down the SW/HW design process, since the allocation is done at the address- and pin-level of accuracy.

Obermaisser et al. [5], [6] propose a novel approach to MPSoC design, by introducing a time-triggered (TT) communication network. Each micro-component of the MPSoC interfaces with the TT interconnect using Trusted Interface SubSystem (TISS), and the TT communication is scheduled and managed by a Trusted Network Authority (TNA) component in accordance with the global clock and global communication time-slot schedule table. Diagnostic and error correction is performed by a dedicated Diagnostic Unit (DU). Such architecture allows determinism and error containment and correction, but has the limitation on scalabity and robustness due to a global clock, which is hard to implement in large systems. Also, the interconnect is restricted to support a single communication protocol managed by the TNA.

Ihmor et al. [2] work on rapid system prototyping of reconfigurable embedded systems.

The developed design flow enables an automated synthesis of interface adapter modules in systems with incompatible SW and HW interfaces. However, the software interfaces are restricted to the form of memory mapped I/O and message based communication is outside their scope. Further, the interface adapter module does not support routing, but only queuing and forwarding functions, from source to (fixed route) destination.

F. Oppenheimer, D. Zhang and W. Nebel published a methodology [3] that allows the automated synthesis of shared memory for the communication of hardware and software via memory mapped I/O. The approach uses an XML based description language (COMIX) that is independent from the target language for modeling hardware/software interfaces. The implemented synthesis tool (COHSID) then automatically generates software device drivers and hardware I/O components from a COMIX specification. However, the input to the synthesis algorithm requires explicit definition of all communication registers.

The work of Zissulescu et al. [9] models and synthesizes point-to-point communication in multiprocessor systems. The synthesis is done in two steps: first includes an automatic generation of a process network with a simple model of a queued (FIFO) inter-process communication. Then, the network is synthesized, by converting the FIFO-based communication into hardware read/write memory operations. This approach focuses only on point-to-point communication, and is not applicable to busses and/or complex Networks-on-Chips systems, because it introduces long delays in the routing process. Further, it only support small data exchanges (scalars) and the usage of large packets instead of scalars in the communication protocol is not feasible.

G. Schirner, R. Doemer [7] propose a System Design Environment for SW development which provides an abstract RTOS model and processor models at 3 level of abstraction. Parts of the design flow are automated so the designer can focus on the algorithm design and space exploration. The approach is useful for ARM, AMBA bus and uC/OS designs, but is not extendible for general platforms with general protocols and RTOS implementations.

Some research effort are focused on raising the level of TLM communication interconnect paradigm from message-blocking FIFO transactions to service based procedures. These services execute over existing FIFO transactions (another comm. layer) and establish the Hardware Procedure Call (HPC) protocol. The HPC transports an arbitrary number of arguments of complex data-types, and it consists of two transactions for service invocation, (input parameters return value length definitions) and one for return value transfer. However, this approach is restricted to proposed HPC protocol, and allows only a restricted set of communication scenarios (simple bus-architecture, no routing, no dynamic scheduling, no service priority etc.) supported by HPC.

3 MPSoC Design

MPSoC design is a process of creating an arbitrary MPSoC platform targeted for an arbitrary application so that the system meets the specified performance requirements. The process consist of two phases. First is (a), the modeling phase, where the design space is searched for the best fit of platform and application, with regards to performance requirements. In this phase the designer references the system's untimed specification model and performance constraints to create the system's TLM with estimated timing and fast simulation. Second is (b), the synthesis phase, where the chosen TLM is implemented. During system synthesis, the designer might reuse certain features of the chosen TLM, but the output PCAM is mostly implemented manually at register transfer level.

We approach the problem of MPSoC design with automatic TLM and PCAM synthesis. The generation of TLM is simplified with GUI-based MPSoC platform definition. The application code from the system's spec is mapped to each TLM component and their communication is made possible by the automatically generated TL channel functions. After TL modeling, the synthesizable PCAM is automatically generated form the TLM chosen for implementation. The PCA communication between the platform components is generated automatically for that particular platform configuration.

The focus of this paper is the synthesis of communication SW performed in the synthesis phase of MPSoC design. It is the part of the Synthesis Tool that inputs the TLM chosen for implementation and outputs the PCAM. The following sections provide an overview of all models in MPSoC design flow and the functionalities of the TLM-to-PCAM Synthesis Tool.

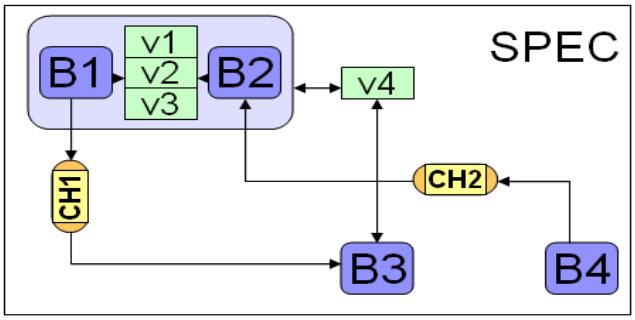

Figure 1: System Spec example.

3.1 Spec Model

The system spec is the untimed, functional application that does not reflect the MPSoC platform or any implementation aspects of the system. The spec code is partitioned into

concurrent processes that exchange data either (a) asynchronously or (b) synchronously. The asynchronous data exchange is enabled with a shared access to a global variable. The synchronous data transfer uses a a dedicated point-to-point channel to synchronize and exchange data between two processes.

Figure 1 shows an example spec. The computations is encapsulated in four processes ($B1$ through $B4$). Additionally, only processes $B1$ and $B2$ have a shared access to variables $v1$ to $v3$, while variable $v4$ is shared among all processes. The asynchronous communication is modeled by links to the global variables, and synchronous communication is presented with channels $CH1$ and $CH2$. The spec model provides validation of the system's algorithmic functionality and serves as a reference model during TLM generation.

3.2 TLM

The TLM is a simulatable model of system computation and communication. The computation is modeled with processes contained in processing elements (or *PEs*), and the communication is modeled with the set of TL channels implementing send, receive, read and write methods. The processes of different PEs (i.e. *remote processes*) communicate with each other via send and receive methods. Read and write methods enable processes to access *shared memory unit*s. The memory accesses of a process do not require synchronization because once the process successfully arbiters for a channel, the memory is guaranteed to be ready for *read/write*. Remote process communication requires both processes to be synchronized prior to any data exchange.

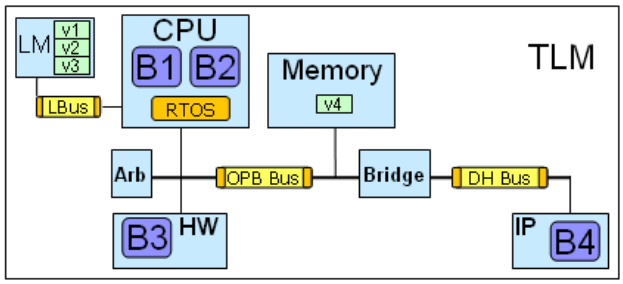

Figure 2: MPSoC TLM example.

Figure 2 shows an example TLM, with three computing components (*CPU*, *HW* and an *IP*) and a shared memory unit, *Memory*. The *CPU* contains local memory *LM* and executes two concurrent processes ($B1$ and $B2$), while *HW* and *IP* each implement a single process: $B3$ and $B4$, respectively. Further, *CPU* and *HW* processes communicate through channel *OPB Bus*. Since they share a channel, an arbiter unit *Arb* resolves possible contentions. *IP*, on the other hand, uses channel *DH Bus*. Therefore, to enable communication between different protocols, a *Bridge* stores and forwards messages between *OPB Bus* and *DH Bus*.

TLMs simulate fast because: (a) the TLM implements both SW and HW in a single, system-level modeling language (e.g. SystemC, SpecC) and (b) the TLM communication is implemented as message transfers with estimated timing and no pin level details in the TL channel. The TL channel contains two event constructs (*req*, *ack*) and a synchronization flag to ensure reliable message transfer between a sender and a receiver process.

3.3 PCAM

Unlike TLM, the PCAM implements communication at the pin-accurate level.
The SW components in PCAM (processors) contain the executable code and communication drivers, compiled and linked with the used libraries. The drivers are processor dependent, written in C/assembly.
The HW components in PCAM (such as IPs and bridge elements) are written using HW description languages (e.g. Verilog, VHDL) at register-transfer level (RTL). Both computation and communication in HW is implemented with finite state machines with data path (FSMD).

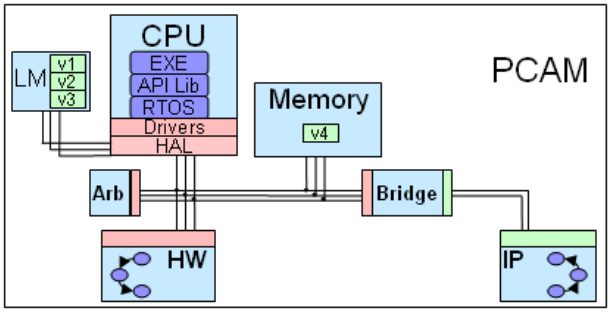

Figure 3: MPSoC PCAM example.

Figure 3 shows the example of a PCAM. As the TLM, it includes a *CPU*, *HW*, *IP*, *Memory*, and arbiter *Arb* and a *Bridge*. The *CPU*, however, contains a compiled executable and an *RTOS* for task scheduling, while *HW* and *IP* implement computation with FSMDs. Moreover, the communication between components is also implemented differently for HW and SW. SW driver for the *CPU* contains interrupt handling routines and data transfer methods. These methods are analogous to the channel methods in the TLM.

The PCAM executes slower then the TLM, but accurately represents the system platform, which allows for an automatic download to FPGA using commercial platform synthesis tools.

4 Synthesis Tool

The synthesis outline is shown on Figure 4. The inputs to the synthesis are the chosen TLM (shown on top of the figure, left) and designer's decisions which are inputted by the designer through the graphical user interface (GUI, on top of the figure, right). For example, the decisions regarding SW communication synthesis include data transfer type, packet size, routing scheme and synchronization mechanism.

The *Parameter Extractor* unit parses through the TLM and identifies PEs and their channel connection(s). Each communicating pair of PEs is then attributed with the parameters describing the their connection. The list of communication parameters is shown in the Table 1. The parameters belong to either of the three protocol classes: process synchronization, routing and data transfer. These protocol classes are described in the following sections.

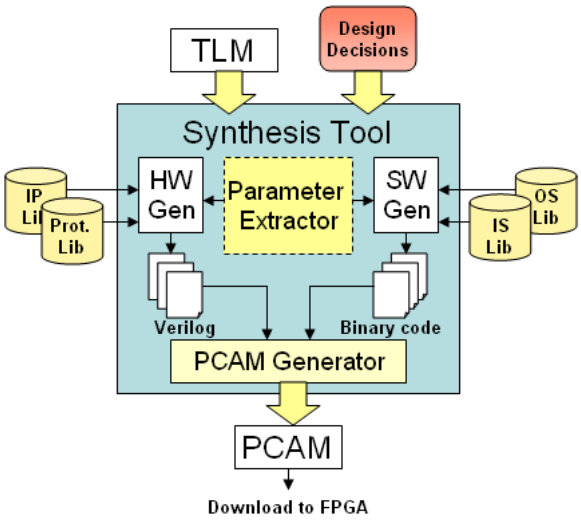

Figure 4: TLM to PCAM MPSoC synthesis.

The *SW Gen*, depending on the parameter values, utilizes different libraries (shown on Figure 4, right) to assemble platform specific drivers for each PE. The synthesized drivers are then compiled and linked to the corresponding application code for each process in the MPSoC that uses them. The *HW Gen* unit applies input parameters to generate bridges for translation of incompatible bus protocols. *HW Gen* is outside the scope of this paper.

The outputs of *SW Gen* and *HW Gen* (Binary and Verilog files, respectively) are fed into the *PCAM Generator*, shown on the bottom of the figure. The generated PCAM can then be automatically downloaded into the FPGA board.

	Parameter Name	Parameter Value(s)
Process Synchronization	**Synch. Flag Location**	In Initiator/Resetter PE
	Synch. Flag Set Style	FIXED/RUNTIME
	Synch. Flag Set Method	SIGNAL/BUS
Routing	**Routing Type**	FIXED/RUNTIME
	Routing Metric	DISTANCE/DELAY
Data Transfer	**Transfer Type**	MSG/PCT
	Packet Size	positive integer
	No. of Bridges	≥ 0

Table 1: List of parameters for communication SW synthesis

4.1 Process Synchronization

Process synchronization is achieved by one process setting the synchronization flag and the other process checking and resetting the flag. The parameters corresponding to process synchronization, as shown in Table 1, are (1) Synchronization Flag Location, (2) Synch. Flag Set Style and (3) Synch. Flag Set Method.

Regarding parameter (1), the synchronization flag may reside in either the PE *Initiator* or in the resetting PE (*Resetter*). The *Initiator* is the PE whose process will write into the synchronization flag first and thus initiate the synchronization. Consequently, the *Resetter* contains the process that will complete the synchronization by resetting the flag.

Parameter (2) defines whether the PEs *Initiator* and the *Resetter* are defined at *compile time* or at *run-time*. If they are defined during compilation, even if the process in the *Resetter* reaches the synchronization point first, it may not set the flag. Instead, the process must read it until it is set by the process in the *Initiator* and then reset it. If the PEs are decided on run-time, the *Initiator* is the PE whose process reaches the synchronization point first.

The flag set/reset method can be implemented with either *polling* or *interrupt*, with regards to Parameter (3). In *polling*, the flag is accessed via bus, while in the *interrupt* scheme, a dedicated (interrupt) signal accesses the flag. The *polling* mechanism contains a *polling* routine with corresponding, bus-addressable *polling register* (i.e. synchronization flag). The interrupt based synchronization contains an *interrupt handler* with its *interrupt flag*. The *interrupt device* must be connected to an interrupt signal to drive.

The figure 5 shows two examples of synchronization implementation. In (a), the flag resides in the local memory of the PE with *Process 1* and the synchronization method is interrupt based. *Process 1* sets/resets the flag with local read/write operations, while *Process 2* sets/resets the flag remotely, with an interrupt signal. Alternately, the flag may reside in the local memory of the *Process 2*, shown on figure (b). Then, *Process 1* regularly reads the flag by polling (test-end-set operation). Every test operation utilizes the bus. (The *flag register* must have a bus address, i.e. be accessible via bus).

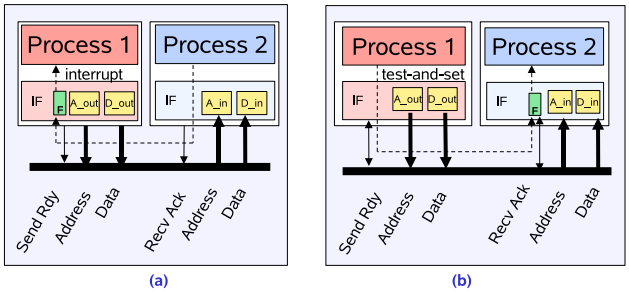

Figure 5: Process Synchronization: (a) Flag in Resetter, interrupt set/reset, (b) Flag in Initiator, polling

4.2 Routing

Routing defines the source-to-destination path through which the data will be transferred. If the source and destination *PEs* are directly connected with the bus, the routing function outputs the unique identifier of that bus. If the *PEs* communicate indirectly, the result of routing function is an ordered string of busses and bridges from source to destination, starting and ending with bus identifiers the source and destination (respectively) are connected to.

$$Route(Process_i, Process_j)) = \begin{cases} \{Bus_k\}, \text{if direct connection} \\ \{Bus_k, B_l, Bus_m, B_n ... Bus_o\}, \\ \text{if bridged connection} \end{cases}$$

The routing parameters shown in Table 1 are: (1) Routing Type and (2) Routing Metric. If the routing function is called during compilation, the routing for each source-destination will be FIXED during execution (Parameter 1). If the process calls the routing function before each send, the routing is decided on RUN-TIME.

Regardless of when it is called, the routing function will output different routes depending on the chosen metric, i.e. the value of Parameter (2). If the criteria for choosing a route is distance, the routing algorithm will take into account the number of bridges and busses connecting the source and destination. Alternatively, the algorithm can output the route with the shortest message transfer delay from source to destination. The time delay of each route can be estimated by inspecting the number of unused FIFO locations in the intermediate bridges. The assumption is that the bridges with the least packet load will transfer the message the fastest.

4.3 Data Transfer

The final communication parameter class is Data Transfer, which contains the following parameters: (1) Transfer Type, (2) Data Packet Size and (3) Number of Bridges. The data that is being exchanged can be transferred either in a single message or packaged into fixed size packets (Parameter 1). For single message transfers, process synchronization is done at the beginning of the transfer and the message is sent in whole. In packet transfers, the sending process packages the message into packets of equal size and synchronizes with the receiving process before each packet transfer.

Parameter (2) is depended on the Transfer Type: message transfers have Packet Size set to 0, while packaged transfers have a positive integer for Packet Size.

Finally, the Number Of Bridges (Parameter 3) has the following effect on data transfer: If there is at least one bridge between the source and the destination, the sending process needs to invoke the service and request FIFO allocation of the first bridge. In direct communication no such requests are needed.

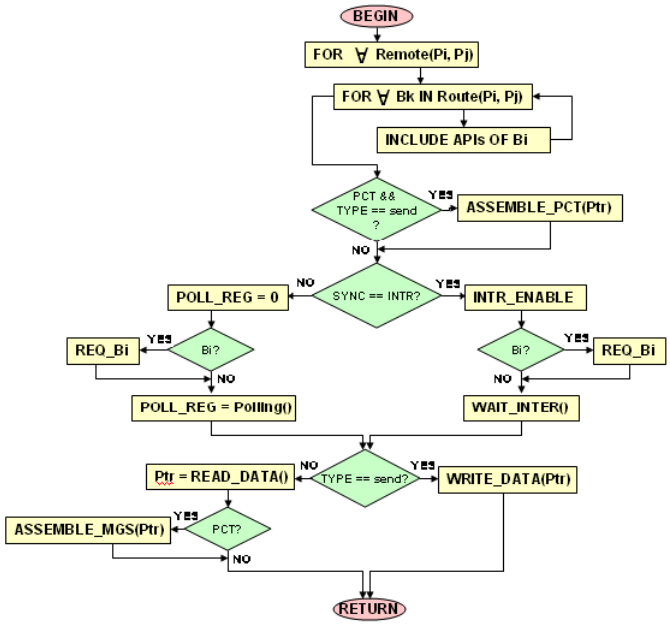

Figure 6: Driver Synthesis Procedure Diagram.

4.4 Synthesis Procedure

At the top level, the communication SW synthesis parses through *PEs* in the system and identifies remote process communication pairs ($comm.type(p_j, p_k)$) For all such pairs, the

driver synthesis generates *PCAM* level drivers to replace the *TL* channel calls with. Figure 6 shows diagram of the partial driver synthesis procedure. For clarity, the diagram captures only the generation of *send/recv* driver methods. The drivers for memory access operations are synthesized similarly, using different values of the synthesis parameter set.

The route determines if the bridge header file will be included in the drivers, and the synchronization *Synch* includes either interrupt enable methods (interrupt based scheme) or polling register initialization (for polling scheme). The synchronization completes either after execution of interrupt handler, or after the polling returns the set $poll\_reg$ value. If the transfer includes message packaging (*PCT*), the message is either disassembled into packets before sending or assembled from received packet after receiving, depending on the type of transfer ($send/recv$).

Finally, all *PEs*, *IPs* and bridges in *MPSoC* are registered as devices in PCAM, with corresponding IF registers. The bridges have one or more request registers, one for each process accessing the bridge. In the interrupt based synchronization, appropriate interrupt flags and interrupt handlers will be registered to it. Similarly, if the device is synchronized with polling, it will have polling registers and polling function.

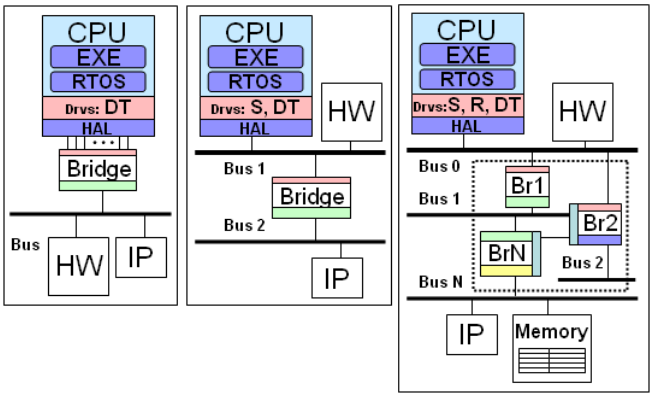

Figure 7: Example of different drivers configurations in MPSoC. Drivers implement (a) data transfer only, (b) process synchronization and data transfer, (c) routing, process synchronization and data transfer

4.5 Bus drivers for PCAM

The output bus drivers are customized to achieve high performance for each design. Figure 7. shows three different driver schemes that can be generated automatically with *SW Gen*. The *CPU* drivers are shown over the hardware abstraction layer (*HAL*) and under the compiled application (*EXE*) code (and possibly an embedded real-time operating system

RTOS). Different properties of custom drivers (shown on figure with symbol *Drvs*) are abstracted with symbols *DT*, *S* and *R*, representing *data transfer*, *synchronization* and *routing* functionalities, respectively.

Figure on the left shows the *CPU* attached to the personalized *Bridge* that interfaces the system bus and handles routing and synchronization with all destination(s), so the drivers need only include the data transfer. Middle figure shows the *CPU* communicating either directly (to the *HW*) or through the *Bridge* (to access the *IP*). Here, the drivers need to implement process synchronization and data transfer, but not routing since the design does not contain multiple routes to any destination PEs. Finally, the right figure shows the scheme in which the *CPU*'s drivers contain all communication features.

5 Experiments and Results

We have applied our approach to an industrial strength example: the MP3 decoding algorithm. The MP3 decoder is a device for decompression of a MP3 input stream that outputs audio data. The input data stream is organized in frames and encoded using the MP3 compression algorithm. The MP3 decoder functionality is shown on Figure 8.

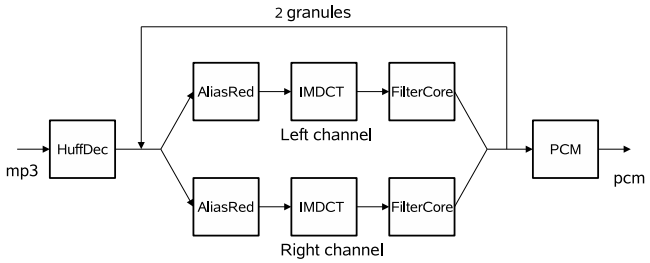

Figure 8: Block functionality of the MP3 Decoder Example.

The first phase uses Huffman tables, (block $HuffDec$), after which each frame is subdivided into two granules of equal size ($Left$ and *Right channel*). Each branch generates requantized output in three steps (blocks): the *AliasRed* for alias reduction, the *IMDCT* block and $FilterCore$ for creating PCM samples. The correctness of our experiments is validated by comparing the generated PCM samples with the reference data stored in the golden file.

5.1 Experiment Setup

Experimental setup includes four different platform configurations of MPSoC implemented on Xilinx Multimedia FPGA Board with configurable Microblaze soft core, running at

27Mhz. The PCAM models of these platform configurations have been implemented both manually and automatically, using our Synthesis Tool.

Each platform contains *Microblaze* processor with 4MB of external memory (*Memory* block) and an *OPB Timer* for timing. The parallelism is introduced with one or more concurrent HW units performing *IMDCT* sampling and/or polyphase filtering *FilterCore* (depending on the design platform). Since HW units communicate with double-handshake (DH) and *Microblaze* processor supports OPB protocol, the *Bridge* is inserted to translate between the two protocols. The data exchanged between the processor and HW units is transferred packets ranging in size from 16 to 36 bytes.

The initial configuration contains only one polyphase filtering (*FilterCore*) HW unit, while the rest of the decoding is running on the *Microblaze*. Next configurtion has two concurrent *FilterCore* units. The third configuration consists of two concurrent *IMDCT* units. Our results show that this configuration yields more speedup over the previous platform designs. Finally, the maximum speedup was achieved with implementing four HW units attached to the *DH* bus. Figure 9 shows the fourth configuration.

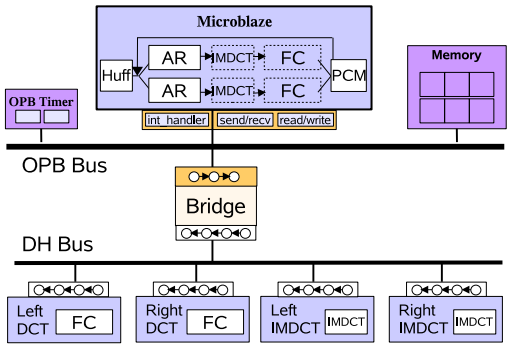

Figure 9: MP3 Decoder Platform: SW + 2 DCT units + 2 IMDCT units.

The transfer of input data from the *Microblaze* to the *IMDCT* units includes 88 data transfers, each having 18 byte packets. The outputs from the *IMDCTs* is transferred back to the *Microblaze* in 88 packets, each 36 bytes in size. Similarly, the input to the *FilterCore* units is sent in 72 data transfers with 16-byte while the output is contained in 144 transfers of 32-byte packets. Each data transfer includes requesting the *Bridge*, synchronizing with the destination with the interrupt scheme and the data transfer. Following are the results of comparisons in communication speed between the manual and automatically generated PCAMs of MP3, mapped on four MPSoC platforms configurations.

	Design	Code size(in bytes) (% diff.)	Total comm. delay (in cycles) (% diff.)	Total comm. delay (in ms)
Manually implemented PCAM	**SW+1DCT**	171,362	957,060	35.45
	SW+2DCT	160,640	1,914,120	70.89
	SW+2IMDCT	163,492	1,875,588	69.46
	SW+2DCT+2IMDCT	153,420	3,789,708	140.36
Automatically generated PCAM	**SW+1DCT**	172,072 (+4.14%)	949,932 (-7.44%)	35.18
	SW+2DCT	161,280 (+3.98%)	1,899,864 (-7.44%)	70.04
	SW+2IMDCT	164,132 (+3.91%)	1,863,972 (-6.19%)	69.04
	SW+2DCT+2IMDCT	153,624 (+1.33%)	3,763,836 (-6.83%)	139.40

Table 2: Comparison of manual vs. automatic PCAMs of the MP3 Decoder

	Design	Code size (in lines) (% diff.)	Development Time (% diff.)
Manually implemented drivers	**SW+1DCT**	162	5 h + 2 h
	SW+2DCT	192	5 h + 2.5 h
	SW+2IMDCT	192	5 h + 2.5 h
	SW+2DCT+2IMDCT	252	5 h + 3.5 h
Automatically generated drivers	**SW+1DCT**	168 (+3.70%)	5 h + 0.14 s (-28%)
	SW+2DCT	208 (+8.33%)	5 h + 0.14 s (-33%)
	SW+2IMDCT	208 (+8.33%)	5 h + 0.14 s (-33%)
	SW+2DCT+2IMDCT	288 (+13.83%)	5 h + 0.14 s (-37%)

Table 3: Comparison of manual vs. automatic bus driver design development

5.2 Results

Table 2. presents the results of our experiments for both models, with measures of design size (in bytes of compiled code for Microblaze processor) and performance (in number of cycles for communication between the processor and the external *HW* units). First we show the measurements for manual designs of all four implemented platforms. As expected, with more external units, the code size of Microblaze processor decreases since the processor is relieved from extra computation (third column), and the communication time increases due to more data transfer operations (shown in columns four, in clock cycles, and five, in milliseconds). The last four rows show results for code size and communication delay of automatically generated models.

The manual code contains a number of optimizations dependent on the MP3 Decoder application. For example, the communication between the *main* process in the *CPU* and the HW units is sequential: for each frame, the *FilterCore* transfers always follow the *IMDCT*

transfers. Therefore, the manually written drivers for the *CPU* all share a single global flag for process synchronization.
However, at this time the *SW Gen* Tool does not implement any static code analysis for similar optimizations, so the automatically synthesized drivers have independent synchronization between unique pairs of processes. This results in occasional redundancies of code in the automatic drivers as compared to the manual design. However, in complex platforms such small code savings provide slim benefits, so automatically synthesized models show comparable results when compared to respective manual designs. The measurements for communication delay (Table 2, columns four and five) show similarity in timing results between automatic design and manual implementations.

Table 3. shows comparison of code size and development time for drivers and communication IFs. Manual design of bus drivers is between 4% and 14% smaller than automatic design (code redundancies).However, the automatically generated bus drivers were synthesized in less than 200 ms. The manual development of bus drivers took approximately 2 to 3 hours to implement, depending on the complexity of the platform configuration. For each platform configuration, these drivers development times have been increased with the time interval spent on design development, i.e. platform configuration design (2-3 hours, depending on the configuration) and testing and debugging (estimated 3 hours). A comparison of manual and automatic design times shows a significant productivity gain of up to 35% of total manual development time, when using our tool.

6 Conclusion

This paper defines a tool for automatic synthesis of MPSoC communication firmware that is customized for its application. The customization is achieved by selecting values for a parameter set that captures its communication protocol(s), and by selecting the appropriate MPSoC platform template. The output code can then be compiled and directly downloaded to the FPGA board.

The major contribution of this paper is, (a) the automation of synthesis, which yields high productivity of MPSoC systems for designers and makes short time-to-market projections realistic to achieve. Our experimental results show dramatic speedup in automatic design implementation since the designer is relieved from error prone coding for drivers of each of the components in MPSoC.

Another benefit of our approach is (b), rapid design space exploration. Also due to automation, there is no need for manual rewriting the code in different code partitioning and mapping schemes. With simple selection of different parameter values, various communication schemes and protocols are synthesized automatically.

Future work includes expanding the parameter set for platform definition to include more complex MPSoC and network-on-chip (NoC) designs. Moreover, we plan to optimize

details of the synthesis algorithm in order to further improve on the size and performance of automatically generated models.

7 Acknowledgments

This work is a continuation of the decades of research in system design and synthesis methodology done by members of the SER group adjoint to the CECS at UCI. We greatly appreciate their valuable inputs. Special acknowledgments go to Hansu Cho for providing the Verilog implementation of Bridge component, and Pramod Chandraiah for the specification model of the MP3 Decoder.

References

[1] A.A. Jerraya, N.E. Zergainoh, AL. Baghdadi, "HW/SW codesign of on-chip communication architecture for application-specific MPSoC," *International Journal of Embedded Systems*, 2005.

[2] S. Ihmor, M. Visarius, and W. Hardt, *Modeling and Automated Synthesis of Reconfigurable Interfaces,* Ph.D. Thesis, Faculty of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn, 2006.

[3] F. Oppenheimer, D. Zhang and W. Nebel, " Modelling Communication Interfaces with ComiX," *Proceedings of the 6th Ade-Europe International Conference*, 2001.

[4] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, A.A. Jerraya, " Flexible and Executable Hardware/Software Interface Modeling For Multiprocessor SoC Design Using SystemC, " *Proceedings of the 12th Asia and South Pacific Design Automation Conference*, 2007.

[5] H. Kopetz et al " Error Containment in the time-triggered SoC Architecture, " *Proceddings of the 2nd IESS*, 2002.

[6] K. Steinhammer, A. Ademaj " HW implementation of the Time-Triggered Ethernet Controller, " *Proceddings of the 2nd IESS*, 2002.

[7] G. Schirner, R. Doemer " Embedded Software Development in a System-Level Design Flow, " *Proceddings of the 2nd IESS*, 2002.

[8] K. K. Ryu, V. J. Mooney III " Automated Bus Generation for Multiprocessor SoC Design, " *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2003

[9] C. Zissulescu, B.Kienhuis, and E. Deprettere, " Laura: Leiden architecture research and exploration tool, " *Proceedings of Design, Automation and Test in Europe Conference*, 2004.

[10] W. Cesrio, A. Baghdadi, L. Gauthier, et al., "Component-Based Design Approach for Multicore SoCs, " *Proceddings of the 39th Design Automation Conference*, 2002.

[11] R. Passerone, J. Rowson, A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *Proceddings of the 35th Design Automation Conference*, 1998.

[12] K.K Ryu, E. Shin, V.J. Mooney, "A Comparison of Five Different Multiprocessor SoC Bus Architectures, " *Proceedings of the EUROMICRO Symposium on Digital System Design*, 2001.

[13] J. Gong, D. Gajski, S. Bakshi, "Model Refinement for HW/SW Codesign, " *Proceedings of European Design and Test Conference*, 1996.

[14] T.Y. Yen, W. Wolf, "Communication Synthesis for Distributed Embedded Systems, " *Proceedings of International Conference on Computer Aided Design*, 1995.

[15] M. Loghi, et al, "Analyzing On-Chip Communication in a MPSoC Environment, " *Proceedings of Design Automation and Test in Europe Conference and Exhibition*, 2004.

[16] S. Narayan, D. Gajski, "Protocol generation for communication channels, " *Proceedings of the 31st Design Automation Conference*, 1994.

[17] S. Narayan, D. Gajski, "Synthesis of System Level Bus Interfaces, " *Proceedings of Design Automation and Test in Europe Conference and Exhibition*, 1994

[18] T. Givargis, F. Vahid, "Parameterized System Design, " *Proceedings of International Conference on Hardware-Software Codesign and System Synthesis*, 2000

[19] L. Benini, G. D. Micheli, "Network on-chips, " *In IEEE Computer vol. 1 pp 70-78*, 2002

[20] P. Guerrier, A. Greiner, "A Generic Architecture for On-Chip Packet-Switched Interconnections, " *Proceedings of Design Automation and Test in Europe Conference and Exhibition*, 2000

[21] D. Gajski et al., "SpecC: Specification Language and Methodology, " Kluwer Academic Publishers, 2000