Automatic Generation of Embedded Communication SW for Heterogeneous MPSoC Platforms

ABSTRACT
This paper propose a tool for automatic synthesis of MPSoC SW communication interface 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 point-to-point channels. Since it does not accurately reflect the MPSoC platform, the TLM cannot be compiled and downloaded to the FPGA board. Our communication synthesis tool automatically transforms TLM point-to-point functions into platform specific IF and drivers in 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 more than 200,000 times productivity gain over manual design, with a performance increase over manually designed drivers between 6% and 8% in speed, while maintaining comparable 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 the ability to execute/compute in parallel. 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.

However, designing MPSoC is complex and time-consuming, making it difficult to meet the stringent time to market constraints. Applying traditional system design methodologies (top-to-bottom, bottom-up) to MPSoC is slow and inefficient due to systems size and the complexity.

Platform based HW/SW co-design is widely seen as a solution to simplifying the design process for MPSoC. The platform based design consists of separating the application code into multiple concurrent processes, mapping each into predefined components in the MPSoC platform and making them communicate correctly. This approach largely outperforms the traditional RTL level (bottom up) design, by raising the level of abstraction to system level. At system level, the MPSoC are described in high-level design languages (SystemC, SpecC, SystemVerilog) and the communication is abstracted with function calls channels. However, such abstracted models cannot be automatically synthesized down to the physical FPGA board because they abstract away pin and cycle accurate details.

This paper proposes a tool for MPSoC design and synthesis that couples the simplicity of high-level, TL modeling style with formal definition of MPSoC platform to achieve automation in synthesis of bus drivers and communication IF for PEs in synthesizable PCAM models. Figure 1. shows the outline of our approach.

![Communication interface synthesis overview.](image)

The tool inputs the TLM of the design, which consists of a set of communicating PEs interconnected with point-to-point channels. An example TLM, shown on the left side of Figure 1, presents one such PE, which communicates with an abstracted set of other PEs in the design with M channels (Ch1 to ChM). The tool also inputs the MPSoC platform template, shown on the top right side of Figure 1. The platform indicates possible routes for each communicating PE. Together with the designer’s decision (top left in Figure 1) regarding the communication implementation (e.g. packet size, synchronization mechanism), the tool combines appropriate commands and method calls to assemble functional bus drivers for each PE in the design.
The tool outputs the synthesizable communication IF for each PE and bus connected to it, along with the corresponding bus drivers. One such output is shown in Figure 1 (right), where the PE interfaces two different buses (Bus1 and Bus2) and uses two different set of bus drivers to drive them. The synthesized pair of IFs is abstracted with two sets of S, R and DT blocks symbolizing the implementation of process synchronization, routing and data transfer (respectively) for each IF.

The rest of the paper is organized as follows: Section 2 outlines the related work. After defining the principle of our tool for automatic communication synthesis, we describe the inputs and outputs of the tool in Section 3. Section 4 presents the necessary tasks included in driver generation, namely process synchronization, routing and data transfer. We present the synthesis procedure in Section 5. Experimentation in Section 6 demonstrates the effectiveness of our tool. We conclude the paper with the summary of contributions in Section 7.

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), which is used as a template throughout their SW/HW design process. This model is represented by an abstract architecture description and an allocation table, and is characterized with modularity, flexibility and scalability. Using GAM-MPSoC and binaries for each processor as inputs, the authors are able to synthesize wrappers for all system components and communication channels and generate a detailed, pin-accurate micro-architecture of the system. However, the process is not fully automatic, since the allocation tables needed for synthesis have to be written manually. Further, each processor dependent communication interface also has to be written manually. Since the network consists of point-to-point connections, the component interfaces can be very complex and include a large number of controllers managing the communication through parallel channels.

Ihmor et al [2] work on rapid system prototyping of reconfigurable embedded systems, and the IP-based System-on-Chip design. The developed design flow enables an automated synthesis of interface adapter modules in systems with incompatible interfaces. The systems comprise of hierarchically structured components for computation and communication. Compatible communication components may directly communicate via the present interconnections, while incompatible tasks get interconnected by automatically generated adapter modules. The synthesized modules cover both hardware and software 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 synthesis is performed on a very low level, since the input to the synthesis algorithm requires explicit definition of all communication registers.

The work of Zissulescu et al [4] 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 buses buses 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.

Figure 2: PCAM synthesis from TLM.

3. PROBLEM DEFINITION: AUTOMATIC COMMUNICATION SYNTHESIS

We approach the problem of meeting short time-to-market deadlines in large and complex multi-processor systems with automatic communication synthesis for MPSoC. If the communication drivers for MPSoC are generated automatically, the designer is relieved from implementing them manually and can focus on other design issues. Also, automation yields speedup of design space exploration, since different communication scenarios are implemented in a matter of seconds.

This paper presents a tool for automatic generation of communication interfaces in PCAM models of MPSoCs. The synthesis outline is shown on Figure 2. The inputs to the synthesis are the event-driven TLM model of the system and a template parameter set that formally describes the MPSoC platform (both shown on the top of the figure). Parameters
are also used to define the details of communication protocol(s) in MPSoC. The values of parameter set are selected through the graphical user interface (GUI, on the figure, left).

Depending on the chosen MPSoC platform and parameter values, the tool will utilize different libraries (shown on Figure 2, right) to aide in the driver synthesis. The tool automatically generates the communication IF and related bus drivers for the output PCAM (bottom of the figure).

3.1 Definition of TLM
The TLM is a high level, executable model of both system computation and communication. The system computation is modeled with the set of processing elements (PE) which contain parallelly executing processes. The processes of different PEs communicate with each other via point-to-point channels. The channels implement send and receive operations between two processes, or read and write operations between a process and a memory unit. Finally, if the processes support different communication protocol, a Bridge component translates messages from one protocol to another.

Figure 3: Example of TLM of MPSoC
Figure 3. shows one TLM example of the MPSoC. The model consists of three computing components (Process A, B and C, mapped to IP, PE 1 and PE 2, respectively) and a shared memory unit, Memory. Units IP and PE 1 support the same communication protocol and exchange their data with function calls to Channel 1. Further, they have access to the same memory unit (Memory), represented with read/write calls to channels Channel 2 and 3, as shown on figure. PE 2, on the other hand, communicates with a different communication protocol. Therefore, to enable communication, a Bridge component translates messages between IP, PE 1 and PE 2.

3.2 Definition of MPSoC platform
The MPSoC platform is captured with a set of parameters fully describing the processing elements, memory units, buses and bridge elements of the platform as well as their interconnection. Figure 4. shows the example of the MPSoC platform consisting of two processors (PE 1 and PE 2) and a hardware unit (IP). The components are connected with two buses (Bus 1 and 2) and a Bridge unit that translates to and from bus protocols of the two buses. Finally, the Memory unit is accessible to PE 1 and IP units directly through Bus 1, and indirectly through the Bridge for PE 2.

3.3 Definition of PCAM
The PCAM contains the executable code, drivers and libraries for all processors in the system. The input application C code is compiled into a binary for the selected processor. Also, the calls to channels connected to the processor are replaced with code for bus divers, written in C/assembly, depending on the chosen processor. The PCAM also contains all the HW components, such as IPs and Bridge elements, as RTL Verilog, generated during HW synthesis. The synthesized drivers for HW and IP components are implemented in RTL, as finite state machines with dataflow (FSMD).

Figure 4: Example of MPSoC platform
Figure 4. shows the example of a synthesizable RTL model. It captures the structure of the MPSoC platform shown on Figure 2, as well as the functionality of the system described with the TLM in Figure 1. Therefore, the PCAM model includes two processor components, PE 1 and PE 2, executing processes Process B and C and a hardware component IP running Process A. The communication between components connected to Bus 1 in enabled with bus drivers Driver 1a, 1b, 1c. Similarly, Bus 2 is driven by Drivers 2a and 2b.

SW drivers for the processor components PE 1 and PE 2 (Driver 1b and Driver 2b) contain C-based interrupt handling routines and data transfer methods which are then compiled for execution. Communication IFs for those processors are defined with necessary IO registers and masks. On the other hand, drivers for HW components, IP and Bridge, are implemented as finite state machines (FSM) in Verilog, and their communication IFs are captured with the list of IO registers, events and signals that trigger them. Note that the Bridge component is abstracted with Drivers 1c and 2a representing message transfers. Other bridge functions, such as message format translation and queueing
Further, each process can store data to the memory unit. The structure of MPSoC is fully captured with the platform template, as defined in Section III.B.

An MPSoC structure can be formally defined as a set of processing elements (PE) and memory units (M) interconnected with a set of buses (Bus) and bridges (B). Their interconnection is represented with the connectivity relation. The structure of MPSoC is fully captured with the platform template, as defined in Section III.B.

The functionality of MPSoC design is distributed among its PEs and represented with a set of concurrent processes, Process, i = 1…n. Processes that belong to different PEs are called remote processes, while the processes of the same PE are local to that PE. Remote processes communicate with each other with message passing routines, exchanging data with send/recv calls to the bus channel. Further, each processes can store data to the shared memory unit, with read/write memory accesses. When a process accesses a memory unit, no synchronization is necessary because the memory unit is always ready for read/write operations. However, remote process communication requires both processes to be synchronized prior to any data exchange.

4. MPSOC IF SYNTHESIS

In order to automatize communication drivers synthesis from the TL model, all TLM components and their interconnection must be formally defined so the model contains no ambiguity. This is pertinent for a correct understanding of the semantics of the design. If the MPSoC design is unequivocally defined, a tool will parse all input information correctly and produce a functioning firmware that is specific to that particular input. Following sections describe inputs to the synthesis as sets of objects and relations between these objects.

If the flag resides in the local memory of the Resetter, the synchronization resembles a typical interrupt based scheme. The figure (1) shows this scheme, with Receiver process as the Initiator of the synchronization. When Sender is ready to send, it either blocks waiting for the setting of the flag, or resumes other, independent computation. A flag is set by the Receiver with an interrupt signal, and the sender can then reset the flag. The synthesis of this scheme includes generating the interrupt logic of the Receiver device that drives the interrupt signal, and the interrupt handler in the Sender that sets/resets the interrupt flag upon interrupt.

Alternately, the flag may reside in the local memory of the Initiator (figure (2), again the Receiver is the Initiator). Then, the Sender regularly reads the flag to test if it is set by polling. Every test operation utilizes the bus. (The flag register must have a bus address, i.e. be accessible to the Sender). After the Receiver sets the flag, the Sender resets it on the first consecutive test operation. This approach is synthesized with implementation of test-and-set (i.e. polling) routine. In polling, the polling routine with corresponding polling register must also be synthesized. Otherwise, in interrupt based synchronization, an interrupt handler, with its interrupt flag must be implemented. Further, the interrupt device must be connected to an interrupt signal to drive.

4.2 Data Transfer

The data that is being exchanged can be transferred either in a single message or packaged into fixed-size packets. For single message transfers, process synchronization is done at the beginning of transfer, and in packet transfers, processes are synchronized before each packet transfer.

4.3 Routing

Routing defines the path from source to destination PE through which the data will be transferred. If the source and destination PEs involved in the data transfer are directly connected with the bus, the routing function outputs the unique identifier of that bus. On the other hand, if the PEs are connected through a set of buses and bridge elements, the result of routing function is an ordered string of buses and bridges from source to destination, starting and ending with bus identifiers the source and destination (respectively) are connected to.

\[ Route(\text{Process}_i, \text{Process}_j) = \begin{cases} \{\text{Bus}_k\}, & \text{if direct connection} \\ \{\text{Bus}_k, B_i, \text{Bus}_m, B_n, \ldots, \text{Bus}_o\}, & \text{if bridged connection} \end{cases} \]

5. SYNTHESIS PROCEDURE

After formal description of MPSoC processes and their relations, namely Route, Synch and Transfer, it is possible to synthesize communication for that MPSoC design. This section presents the synthesis procedure for the communication drivers of MPSoC.

Procedure 1 parses through the PE objects in the MPSoC design one at a time:

\[ \text{PE}_i, i = 1 \ldots n \]
For each PE, the procedure identifies the processes \( p_j \) \((j = 1 \ldots m)\) that are part of a remote process communication pair \( \text{comm.type}(p_j, p_k) \) (Lines 4-5).

Each communication can be typed \( \text{send} \) or \( \text{recv} \):

\[
\text{comm.type}(p_j, p_k) = \text{send/recv}
\]

For all such pairs, the driver synthesis (Procedure 2) is called (Lines 6-7). Procedure 1 then performs device registration for all PEs, IPs and bridges in MPSoC (Lines 10-18). Device registration is outlined in Procedure 3.

The driver synthesis in Procedure 2 will generate PCAM level driver methods which will replace all TLM function calls for communication. For clarity, the procedure only captures the generation of driver methods for remote process communication (i.e. \( \text{send/recv} \) functions in TLM). The drivers for memory access operations are synthesized similarly, using different values of the synthesis parameter set.

Using the parameter values that capture the input MPSoC platform, (i.e. \( \text{Route, Synch and Transfer} \)) relations the tool generates processor commands which assemble bus driver method. More precisely, \( \text{Route} \) relations determines which header file will be included (Lines 1-3 include a header for bridge \( b_i \) that is on route of \( \text{comm}(p_j, p_k) \)). Also, if there exists \( b_i \), the process needs to send the request for its services (Lines 14-16). Depending on the chosen \( \text{Synch} \) scheme, the drivers first initialize the synchronization process with enabling interrupts for PEi (interrupt based scheme), or setting it’s polling register \( \text{poll.reg} \) to zero (for polling scheme) (Lines 9-13). The synchronization is complete either when the interrupt has been received and handled, or when the polling function returns the set \( \text{poll.reg} \) value (Lines 17-22). Finally, if \( \text{Transfer} \) includes message packaging, the message is either disassembled into packets before sending (lines 6-8), or assembled from received packet after receiving (lines 26-28), depending on the type of transfer (\( \text{send/recv} \)).

Procedure 3 generates files with IF registers for each device. All bridges will have one or more request registers, one for each process accessing the bridge (Lines 3-4). If the device includes interrupt based synchronization, appropriate interrupt flags and interrupt handlers will be registered to it (Lines 6-9). Similarly, if the device is synchronized with polling, it will have polling registers and polling function (Lines 10-14).

After driver synthesis, the platform specific compiler compiles the executable files for the processor cores, and HW synthesis tool generate IP components. The design can then be executed on the FPGA board reflecting the MPSoC platform.

### 6. EXPERIMENTS AND RESULTS

To test our synthesis tool for correctness and efficiency, 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. Following is a short description of the MP3 decoder functionality, as shown on Figure 7.

The first phase of decoding algorithm uses Huffman tables, represented with block \( \text{Huff Dec} \) in Fig. 7. After Huffman decoding, each frame is subdivided into two granules of equal size. From then, each granula can be processes independently, which allows the allocation of two parallel computing
Procedure 3 RegisterDevice(device)
1: device = PE, bridge or IP component
2: gen: DefineIOReg(device);
3: if device = bridge then
4: gen: DefineReqReg(device);
5: end if
6: if SynchMechanism(device) = INTRP then
7: gen: RegisterIntrHandler(device);
8: else
9: // Else, polling based synch
10: gen: DefinePollFunction(device);
11: gen: DefinePollReg(device);
12: end if

Figure 7: Block functionality of the MP3 Decoder Example.

components to achieve speedup. This is shown on Figure 7, with two equal branches (Left and Right channel).

Each branch converts the Huffman symbols to the original, requantized output data in three steps (blocks). First, the AliasRed block is used to reduce aliasing of the encoded data, which is then fed into the IMDCT block. Finally, the polyphase filter block (FilterCore) transforms the data into PCM samples. The PCM samples can now be fed to a loudspeaker or any other output device through appropriate interface. In our experiments, the correctness of the execution is validated by comparing the PCM samples with the reference data stored in the golden file.

6.1 Experiment Setup
Initially we used a TL model of the MP3 Decoder as reference to both manual implementation and automatically generated PCA models. Figure 8 presents the referenced TLM.

Figure 8: Block functionality of the MP3 Decoder Example.

It includes a processing element (PE) containing process

MP3_Main and four IP components, with processes Left_filter, Right_filter, Left_PCM and Right_PCM. Process Main_MP3 accesses input data from the Memory unit, performs Huffman decoding and alias reduction. Then, it distributes the data to each IP unit for IMDCT computation and filtering. Finally, it collects the output data and compares it to the golden file data for accuracy check.

Using this TLM, we have manually implemented four different pin-accurate, cycle-accurate models of the MP3 Decoder using Xilinx Multimedia FPGA Board with configurable Microblaze soft core, running at 27Mhz. The same platform templates have been used as inputs to our synthesis tool, to generate equivalent PCAMs.

Figure 9: MP3 Decoder Platform: SW + 1 DCT unit.

Each platform contains Microblaze processor that uses 4MB of external memory (Memory block) and an OPB Timer for timing the processor communication. The parallelism is introduced into the design 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, a Bridge unit is inserted to translate between the two protocols. The data exchanged between the soft core and HW units is transferred packets ranging in size from 16 to 36 bytes on the design platform.

The initial platform, shown on Figure 8, contains only one concurrency IP component that performs polyphase filtering (FilterCore), shown with shaded block in Microblaze PE, on Figure 8. The rest of the MP3 decoding algorithm is running on Microblaze processor.

Figure 8 shows the platform in which two DCT filters perform polyphase filtering concurrently.

Alternatively, Figure 9 shows two HW units executing IMDCT blocks in parallel. Our results show that this configuration yields more speedup over the previous platform design.

Finally, the maximum speedup was achieved with implementing four HW components attached to the DH bus. Figure 10 shows this configuration.

All PCA models have been compiled and downloaded to the
<table>
<thead>
<tr>
<th>Design</th>
<th>Code size (in bytes) (+/- % difference)</th>
<th>Total communication delay (in cycles) (+/- % difference)</th>
<th>Total comm. delay (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually implemented PCAM</td>
<td>SW + 1 DCT</td>
<td>171,362</td>
<td>957,060</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT</td>
<td>160,640</td>
<td>1,914,120</td>
</tr>
<tr>
<td></td>
<td>SW + 2 IMDCT</td>
<td>163,492</td>
<td>1,875,588</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT + 2 IMDCT</td>
<td>153,624</td>
<td>3,789,708</td>
</tr>
<tr>
<td>Automatically generated PCAM</td>
<td>SW + 1 DCT</td>
<td>172,072 (+4.14%)</td>
<td>949,932 (-7.44%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT</td>
<td>161,280 (+3.98%)</td>
<td>1,899,864 (-7.44%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 IMDCT</td>
<td>164,132 (+3.91%)</td>
<td>1,863,972 (-6.19%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT + 2 IMDCT</td>
<td>153,420 (-1.33%)</td>
<td>3,763,836 (-6.83%)</td>
</tr>
</tbody>
</table>

Table 1: Comparison of manually and automatically generated PCAM models of the MP3 Decoder

<table>
<thead>
<tr>
<th>Design</th>
<th>Driver code size (in lines of code) (+/- % diff.)</th>
<th>Development Time (+/- diff.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually implemented drivers</td>
<td>SW + 1 DCT</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT</td>
<td>192</td>
</tr>
<tr>
<td></td>
<td>SW + 2 IMDCT</td>
<td>192</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT + 2 IMDCT</td>
<td>252</td>
</tr>
<tr>
<td>Automatically generated drivers</td>
<td>SW + 1 DCT</td>
<td>168 (+3.70%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT</td>
<td>208 (+8.33%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 IMDCT</td>
<td>208 (+8.33%)</td>
</tr>
<tr>
<td></td>
<td>SW + 2 DCT + 2 IMDCT</td>
<td>288 (+13.83%)</td>
</tr>
</tbody>
</table>

Table 2: Comparison of manual vs. automatic bus driver design development

Xilinx FPGA board. The MP3 application has then been set to run, with the OPB Timer unit counting the clock cycles of each data transfer between the Microblaze and the Bridge unit.

The HW components communicating with the processor are FilterCore and IMDCT units, executing DCT and IMDCT decoding. In platforms containing FilterCode units implemented in HW, the filtering process includes 72 data transfers between the processor to the unit, in 16 byte packets (DCT send driver). The decoded data is then received from the FilterCore in 144 packets, each 32 bytes in size. Similarly, in platforms with IMDCT HW units, 88 data transfers with 18-byte packets have been sent, and 88 transfers of 36-byte packets with decoded data have been received by the processor during MP3 execution.

Each measured data transfer includes requesting the Bridge unit, waiting for the interrupt signal from the Bridge, indicating the receiving process is ready for transfer, and data transfer which takes place after interrupt handling. Following are the results of comparisons in communication speed between the manual and automatically generated PCA models of MP3, mapped on four MPSoC platforms of different sizes and communication complexity.

6.2 Results

Comparisons between the MP3 PCAM model synthesized with our tool and one implemented manually disclose fully the vast benefits of our tool over manual design. Table 1 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 speed of 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 communication...
The output code can then be compiled and directly downloaded to the FPGA board. The customization is achieved by selecting values for appropriate MPSoC platform templates.

The contributions of this paper are multifold. First, the 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 contributions of this paper are multifaceted. First, the automation of synthesis yields high productivity of MPSoC systems for designers, making short time-to-market projections realistic to achieve. Our experimental results show dramatic speedup in design implementation with the use of our tool, since automation relieves the designer from error prone bus drivers coding for each of the components in MPSoC. Also due to automation, there is no need for manual rewriting the code during design space exploration. Different code partitioning and mapping schemes are automatically taken into account with the next cycle of communication synthesis. Next, protocol customization ensures an easy and error resilient search for the best fit of communication infrastructure for both the application and MPSoC architecture. 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 the size and performance results of automatically generated models.

8. REFERENCES


