Modeling Kahn Process Networks on MPSoC Platforms

Ines Viskic, Daniel Gajski

Technical Report CECS-08-08
July 11th, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

iviskic@uci.edu , gajski@ics.uci.edu

Abstract

Kahn Process Network (KPN) is a widely accepted model of computation in system application design. KPN consist of concurrent processes communicating with unidirectional FIFO channels (Kahn Channels). KPN processes are self scheduled and dependent only on the input data stream, which makes them particularly suitable for media processing and other data streaming applications.

On the other hand, to shorten time-to-market projections, embedded systems designers are designing complex Multi-Processor Systems on Chip (MPSoCs) by using templates of system platforms and reconfiguring predefined MPSoC components. The design process separates the application into multiple concurrent processes and maps each onto a platform component.

However, FIFO channels in the KPN specification are point-to-point primitives and cannot be directly mapped to the bus centric MPSoC platform. Therefore, implementing KPN applications on MPSoC requires manual application recoding and/or platform implementation. This paper presents a simple and efficient approach to synthesize KPN on MPSoC platforms. It proposes a 2-phase adaptation of the Kahn Channel communication of KPN to reflect the underlying MPSoC platform. The first phase replaces Kahn Channels with a memory segment and two point-to-point channels. The second phase maps each memory segment onto the memory component and each of the two channels into the bus components of the MPSoC platform.

Content:

1. Related Work **3**
2. Kahn Process Networks **4**
3. Platform Based Design **4**
3.1. Inputs 4
3.2. Output 4
4. Problem Definition **5**
5. Solution **6**
6. Phase 1: Kahn Channel Mappings to Spec Channels + Storage Elements **6**
6.1 Shared Memory implementation 7
6.2 Shared Tx unit implementation 8
7. Integration of KPN to ESE Modeling Tool (Front End) **9**
7.1. Proposed extensions to GUI/System Capture 9
7.2. Proposed extensions to TLMgen 10
7.3. Proposed extensions to TLMest 13
8. Phase 2: Spec Model Mapping to MPSoC Platform **13**
9. Experimental setup **14**
9.1. Platform 1 15
9.2. Platform 2 15
9.3. Platform 3 16
9.4.Communication profile of H264 Encoder 16
9.5. Results **17**
10. Conclusion **18**
References **19**

List of Figures:

1. An example process network model **6**
2. KPN in ESE Design Flow **7**
3. Proposed Adaptation of KPN to support Platform Based Design. **8**
4. Kahn FIFO mapping to indirect communication paradigm:
(a) via shared memory and (b) via bridge (TX) unit. **9**
5. Shared TX unit implementation of Kahn Channel **9**
6. Shared Memory implementation of Kahn Channel **10**
7. ESE Modeling Tool **11**
8. GUI window for Kahn Channel definition **12**
9. FlagStruct implementation (in *ubc.sc* file) **12**
10. Implementation of *read_mem_flag* method (in *ubc.sc* file) **13**
11. Implementation of *write_mem_flag* method implementation (in *ubc.sc* file) **13**
12. Memory Module definition (in *tlm.cpp* file) **14**
13. Send/Recv communication methods (in *tlm.cpp* file) **14**
14. Time delays definition (in *ubc.sc* file) **15**
15. Spec Channel Mapping into the UBC object **15**
16. H.264 Application Specification **16**
17. MPSoC Platform1 . . . . . **17**
18. MPSoC Platform2 . . . . . **18**
19. MPSoC Platform3 . . . . . **18**

List of Tables:

1. Properties of KPN channel. **6**
2. Properties of Shared Memory implementation of Kahn Channel **10**
3. Properties of shared TX unit implementation of Kahn Channel **10**
4. Communication profile of the H264 Encoder mapped to Platform1 to Platform3 . . **19**
5. Simulation time of TLM models of the H264 Encoder mapped to Platform1 to Platform3 . . . . . **19**

05/13/2008
Ines Viskic

Modeling Kahn Process Networks on MPSoC Platforms

Abstract

Kahn Process Network (KPN) [4], [5] is a widely accepted model of computation in system application design. KPN consist of concurrent processes communicating with unidirectional FIFO channels (Kahn Channels). KPN processes are self scheduled and dependent only on the input data stream, which makes them particularly suitable for media processing and data streaming applications.

On the other hand, to shorten time-to-market projections, embedded systems designers are designing complex Multi-Processor Systems on Chip (MPSoCs) by using templates of system platforms and reconfiguring predefined MPSoC components. The design process separates the application into multiple concurrent processes and maps each onto a platform component.

However, FIFO channels in the KPN specification are point-to-point primitives and cannot be directly mapped to the bus centric MPSoC platform. Therefore, implementing KPN applications on MPSoC requires manual application recoding and/or platform implementation. This paper presents a simple and efficient approach to synthesize KPN on MPSoC platforms. It proposes a 2-phase adaptation of the Kahn Channel communication of KPN to reflect the underlying MPSoC platform. The first phase replaces Kahn Channels with a memory segment and two point-to-point channels. The second phase maps each memory segment onto the memory component and each of the two channels into the bus components of the MPSoC platform.

1. Related Work

As MPSoCs become larger and more complex, platform based modeling is seen as an efficient methodology that improves design productivity. METROPOLIS [6] is a three phase simulation model for platform based design of heterogeneous MPSoCs. Kopetz [7], [8] proposes a component model for dependable automotive systems. Both approaches aim to achieve dependability and reliability of heterogeneous MPSoC by modeling systems using predefined platform templates. However, their design flow requires input models that are compatible with the given platform. The Eclipse architecture [2] offers a platform template for heterogeneous media processing systems. It focuses on bus centric platforms with shared on-chip memory communication, while our work supports both same protocol communication (via memory) and communication of incompatible bus protocols (via transducers).

Several approaches work on implementing application models such as KPN to MPSoC platforms. ESPAM tool [1] inputs KPN application model, an abstract platform description and an application-to-platform mapping to automatically generate communication components and drivers. However, the supported platforms are limited to processors with local memories for local write operation and remote read access to the local memory of every other processor in the platform. Furthermore, all processors need to support the same communication protocol in order to read each others local memories. Our approach supports more general communication architecture of MPSoC.

In [3], [9], the goal is to model the system at transaction level with both purely functional and estimated timing for performace estimation. The inputs are application and platform specifications along with the mapping decisions. The requirement in [3] is that the input application complies with a communication architecture consisting of point-to-point blocking channels with no storage capabilities. We support application models with both blocking channels as well as FIFO channels with bounded storage capacities.

2. Kahn Process Networks

Kahn Process Network (KPN) is an efficient programming model for specification of high-performance, data-dependent media processing applications. KPN are especially effective in signal processing and data streaming, since the functional behavior of KPN is determined only by the input data, with no regard to the order in which the Kahn processes are executed. KPN processes perform their computation concurrently on their private state space, and the communication is done through unidirectional point-to-point FIFO channels.
In a theoretical model, the FIFO channels have unbounded capacity and are accessed with blocking read and non-blocking write actions.

A simple example of KPN is shown in Figure 1, with 3 processes (P1, P2 and P3) connected with 2 FIFO units (f1 and f2) with sizes m, n > 1. In P1→P2 communication, sender P1 "pushes" the data in the FIFO f1 with the non-blocking *write*() function call and receiver P2 pulls it from the same FIFO with the call *read*().In P2→P3 communication, P2 will write in FIFO f2 and P3 will read from it. If the receivers P2 and P3 call *read*() while their FIFOs are empty, they will block until the data is stored in the corresponding FIFO by sending process(es) P1 and P2, respectively.

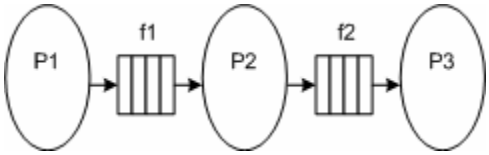


Figure 1. An example process network model.

Table 1 lists the properties of KPN channels, with columns for the benefits (PROs) and the drawbacks (CONs) of such approach.

PROs	**CONs**
1. Literal implementation of KPN model of computation	1. FIFO channel not reflective of the platform
2. Process synchronization done via FIFO accesses Non-blocking write = push in FIFO Blocking read = pull from FIFO	2. FIFO is type-defined, so messages with different types need to be either unified into a single type (e.g. byte) or transferred via separate FIFOs

Table 1. Properties of KPN channel

3. Platform Based Design

Implementing systems on chip (SoC) has traditionally assumed describing the system at the RT (register transfer) level using a hardware description language such as VHDL or Verilog. With the advance of Multi-Processor SoCs (MPSoC) and rising complexity of system applications, the abstraction level for specifying systems has moved above RTL.

The emerging platform based design speeds up the design process by supporting component reuse and by adopting communication paradigms that hide pin accurate details from the designer. It enables specifying both HW and SW aspects of the system at the same time, and consists of separating the application code into multiple concurrent processes and mapping each into predefined components in the MPSoC platform. The design process inputs the specification model (Spec) of the application and the MPSoC platform on which the application will be mapped. The output is a functional transaction level model (TLM) of the system, used to evaluate

the system's performance with regard to computation/communication time-delay, buffer/memory size requirements and bus utilization.

4.1. Inputs

Specification model (Spec) is the functional representation of the system, with concurrent processes encapsulating computation and communicating through bus channels. The channels implement send/recv/write/read function calls with the communication primitives for routing, synchronization and data transfer.

MPSoC platform is a net-list of platform components used for implementing computation, communication and storage capacities (memories). Computing components are processor cores, ASIC units and HW accelerators, while buses, bridges and interface units implement communication.

4.2. Output

TLM models computation with processes contained in modules, while the bus communication is modeled with the set of Universal Bus Channels (UBCs). The UBC represents the system bus(es) and implements send/receive methods and shared memory access.

4. Problem Definition

The communication of KPN is implemented with read and writes to unidirectional FIFO channels. However, FIFO channels are point-to-point primitives and as such cannot be directly mapped to the platform. This is so because the connectivity of the components of MPSoC platform is based on the shared system bus instead of point-to-point links.

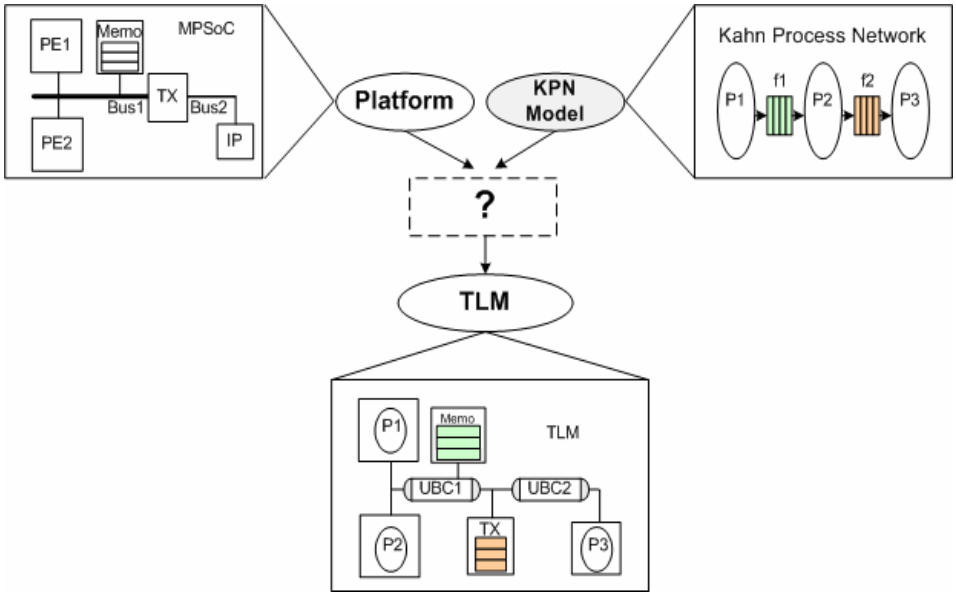


Figure 2. KPN in ESE Design Flow.

Therefore, in implementing the KPN system, the designer is burdened with a task of manually partitioning and mapping the application with regards to the chosen MPSoC platform. The issue is shown on Figure 2.

5. Solution

We propose a simple, 2-phase method of adapting the FIFO communication of KPN to reflect the underlying MPSoC platform. In the first phase, each FIFO Kahn channel is broken down to a memory fragment and two point-to-point channels providing access to it to the corresponding Kahn process. The sender process will access the memory fragment with the write primitive of the first channel, and the receiver will read the memory using the second channel. The second phase maps each memory fragment onto the memory component and each of the two channels into the UBC components of the MPSoC platform. KPN processes are directly mapped to Modules of the platforms (Modules can contain >1 processes).
Our approach is shown on figure 3.

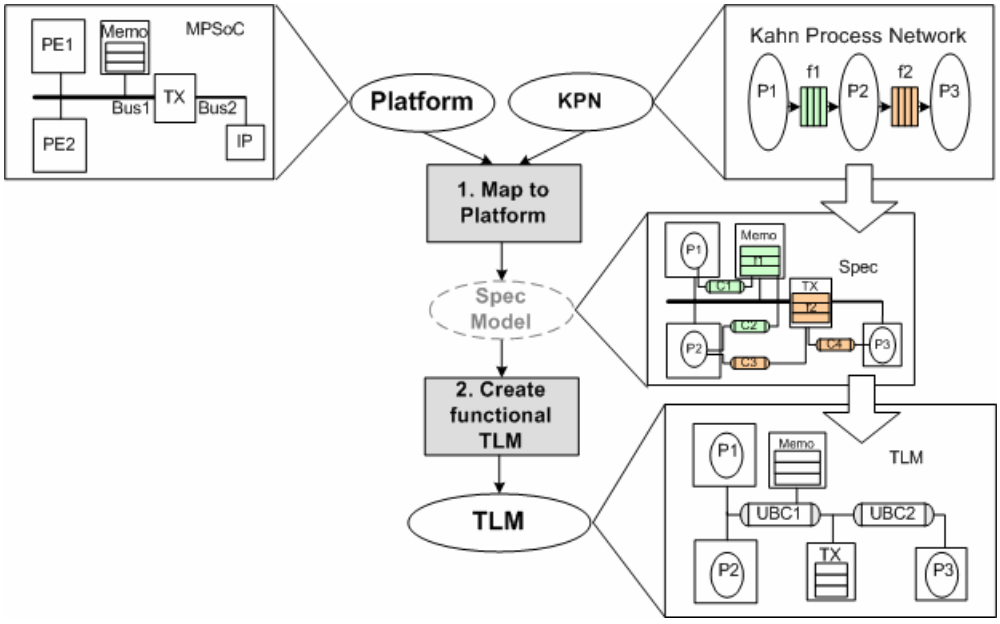


Figure 3. Proposed Adaptation of KPN to support Platform Based Design.

6. Phase 1: Kahn Channel Mappings to Spec Channels + Storage Elements

In our 2-step approach, each Kahn Channel is first replaced with two double-handshake Spec Channel(s) and a storage element (memory, bridge). The Kahn Channels will be transformed into the following objects:

1. 2 Spec Channels + shared memory fragment
    a. Shared memory fragment is **local** to both processes
    b. Shared memory fragment is **global** to both processes
2. 2 Spec Channels + shared TX unit

Shared local memory fragment can be used as comm. buffer only if both processes reside on the same processor/Module (intra-processor communication). If processes are remote but both support the same communication protocol, they can use **global shared memory fragment** as a communication buffer (inter-processor communication). **TX unit** is needed for protocol translation when one process is in a module that resides on a different bus (and supports different communication protocol) than the other process.

Regardless of the type of communication buffer (memory fragment or TX), two channels C1 and C2 implement read/write accesses to it. The non-blocking write property of a Kahn channel is preserved, since the sender process (P1) can resume execution after writing into the memory regardless of the state of the receiver (P2). As in KPN, the receiver will not resume its execution until the data is read (blocking read property).

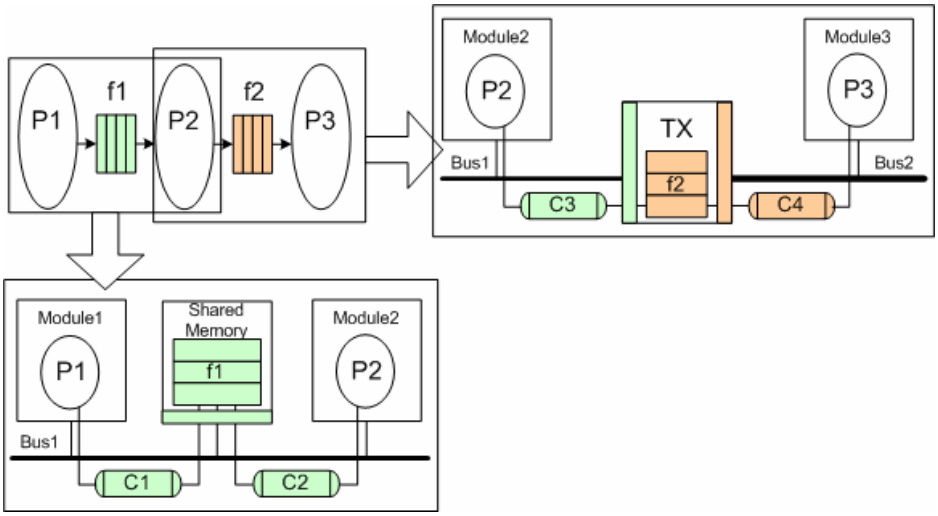


Figure 4. Kahn FIFO mapping to indirect communication paradigm:
(a) via shared memory or (b) via bridge (TX) unit

Figure 4. shows two possible mappings of a Kahn channel to an inter-processor communication architecture, with each process residing in different modules and sharing a global memory. If the modules of sending and receiving processes reside on the same bus, channels C1 and C2 will implement the same communication protocol (C1 and C2 highlighted green, in Figures 4 below). However, if processes support different protocols, they must use the services of the bridge unit TX in order to communicate (Figure 4. right). The TX unit will accept the data from the sending process using one protocol (C3, highlighted green) and forward it to the receiving process with the other protocol (C4, highlighted red)

Finally, if the processes reside in the same module (intra-processor communication), the shared memory is local to that processor. In such case, processes do not use the bus to access the memory, but instead call the RTOS. As before, the RTOS communication services are modeled with a point-to-point blocking channel to the local memory.

6.2. Shared Memory implementation

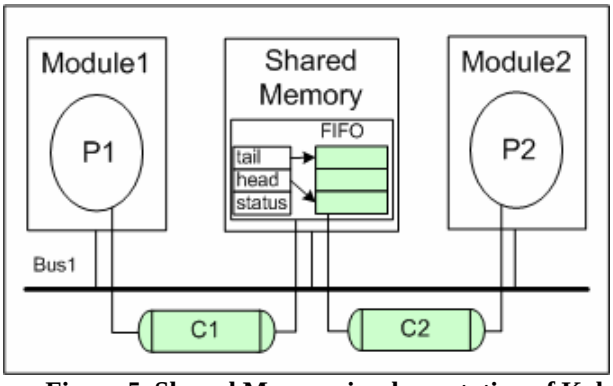


Figure 5. Shared Memory implementation of Kahn Channel

Figure 5. shows one shared memory implementation of a FIFO, with a memory array (FIFO) and three integer variables: a pointer *head* indicating the first address location available for write access, a pointer *tail* pointing to a first available read access location and a *status* flag that annotates the number of currently stored bytes.

Both processes must test the *status* flag before writing to or reading from the location indicated with *head* or *tail* pointer, respectively. Testing the flag is performed within the communication routines (*send/recv*) implemented in the channels C1 and C2. If the memory is full, the sending process will wait for the POLLING_INTERVAL before polling again. It will not leave the channel call *send*() until the message is stored in the memory. On the other hand, if the memory is empty, process will poll the memory every POLLING_INTERVAL (within the *recv* call) until the message is available.

PROs	**CONs**
1. Reflects the implementation model of the platform better than the input KPN model	1. Requires (minimal) changes to current UBC
2. In complex systems with n>1 FIFO units, there can be different options to map FIFO into m <> n Memory Modules	2. Memory is untyped, so messages need to be converted into byte stream
	3. Internal fragmentation

Table 2. Properties of shared memory implementation of Kahn Channel

6.3. Shared TX unit implementation:

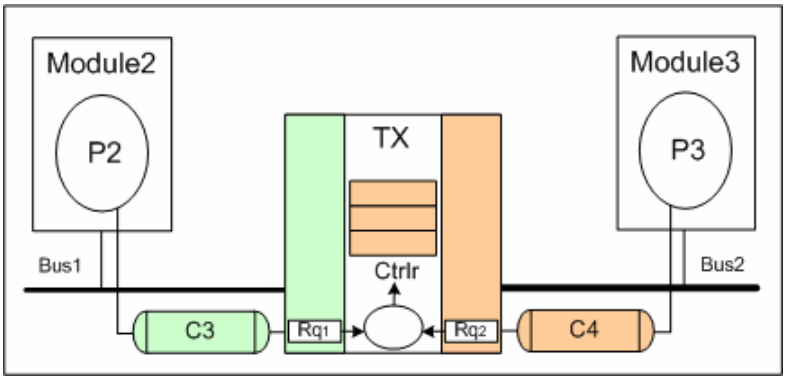


Figure 6. Shared TX unit implementation of Kahn Channel

Kahn channel mapping that includes a bridge unit TX (Figure 6) has channels with different protocols (C3 and C4). Each process uses its channel to first request the service of the TX and than transfer the message to the FIFO in the TX. The service requests are stored in the corresponding registers *Rq1* or *Rq2*, depending on the process that called it. The incoming TX requests and the access to the FIFO are managed by the TX controller process (*Ctrlr*, as shown on Figure 6). If the requests cannot be immediately serviced, the process that called it will block until the controller can accommodate it.

PROs	**CONs**
1. Reflects the system platform	1. Redundant use of area and memory of TX
2. With n>1 FIFO units, there can be different options to map FIFO into m <> n TX units	2. Redundant code to communication: need to send request to TX before actual data transfer
3. Requires no changes to current UBC	3. Overhead in processing data transfer, due to TX

Table 3. Properties of shared TX unit implementation of Kahn Channel

7. Integration of KPN to ESE Modeling Tool (Front End)

Figure 7. outlines the ESE Modeling Tool, that inputs the Spec model of the system and outputs its functional TLM (with no timing annotations) and TLM with estimated timing. The Tool allows the system designer to define the MPSoC Platform through the interactive *GUI* and then to map the application source code to the platform within the *System Capture*. The captured system is then input into the TLM generator to produce the SystemC TLM, composed of 3 main files:

1. *tlm.cpp* – defining the platform and the application mapping in SystemC
2. *ubc.sc* – defining the bus model(s) and the system communication
3. *tx.sc* – defining the bridge model(s) (if any exist in the system)

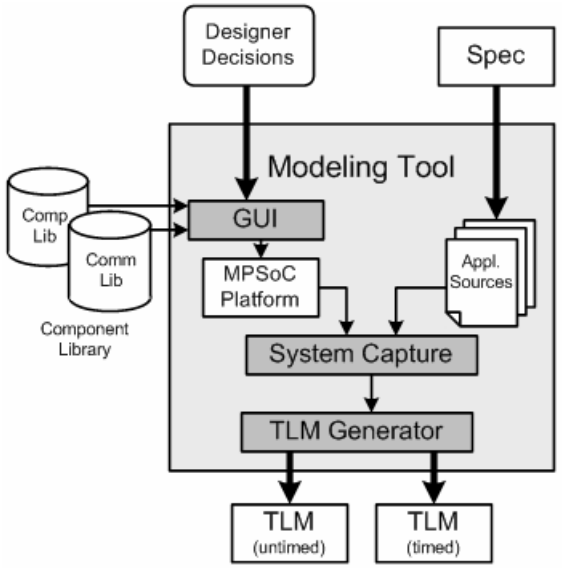


Figure 7. ESE Modeling Tool

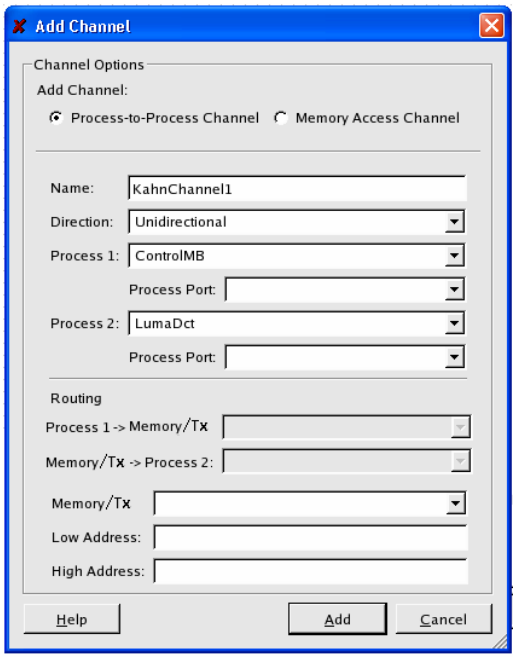
In order to integrate KPN into the Embedded System Environment (ESE) Tool, following needs to be incorporated into the ESE:

7.1. Proposed extensions to GUI/System Capture

The designer defining the Spec Model of the system needs to be able to create a Kahn Channel. A Kahn Channel is a unidirectional channel with buffer capacity and a non-blocking write property. It is fully defined with:

1. Process Sender ID/interface: e.g. *send_P_ID_Sender_P_ID_Receiver*
2. Process Receiver ID/interface: e.g. *recv_P_ID_Receiver_P_ID_Sender*
3. Shared memory fragment or a TX unit, along with:
    a. corresponding routes from each process to the memory fragment/TX
    b. FIFO size (i.e. address range for a memory fragment or TX FIFO)

Therefore, the window menu for Kahn Channel definition would be very similar to the definition of a Memory Channel except that, instead of connecting a process with a memory



connects two processes via an intermediary memory fragment. Also, unlike the Memory Channel, the Kahn Channel communication methods (e.g. *send_P_ID_Sender_P_ID_Receiver, recv_P_ID_Receiver_P_ID_Sender*) would include FIFO management methods (*read_flag, write_flag*).

An example of a GUI window for Kahn Channel definition is shown on Figure 8.

Figure 8. GUI window for Kahn Channel definition

7.2. Proposed extensions to TLMgen

Incorporating KPN into TLM generation (TLMgen) does not require any changes to the existing TX model (the tx.sc will remain unchanged). However, for mapping the Kahn Channel into 2 channels + shared memory fragment, TLM generator needs to implement a memory fragment with structures and methods of managing a circular FIFO. As stated in Section 6.2., a FIFO is defined with a memory array and 3 unsigned integer flags: *fifoValue, fifoHead* and *fifoTail:* the first contains the number of bytes currently stored in the FIFO, the second two are private pointers to the beginning and end of the FIFO. The pointers are updated upon every read and write to the memory.

```
typedef struct flag_structure {
2.       unsigned int Value;
3.       unsigned int fifoHead;
4.       unsigned int fifoTail;
5. } FlagStruct;
```

Figure 9. FlagStruct implementation (in *ubc.sc* file)

Figure 9. shows the set of flags needed for FIFO management (implemented as a structure *FlagStruct*) in the UBC definition file (*ubc.sc*). Figure 10. and 11. present the *read_mem_flag* and *write_mem_flag* methods, respectively, that are also stored in the *ubc.sc* and invoked on every access to the memory fragment.

The *read_mem_flag* method is invoked before the read/write data transfer to assure that the valid data exists in the fragment (Figure 10. lines 13-16, for UBC_READ) or that there is enough free space to write the new data (Figure 10. lines 17-20, for UBC_WRITE).

```
 1. extern "C" void read_mem_flag(unsigned int ProcID, unsigned int MemFlagAddr,
 2.           FlagStruct *Flag, unsigned int MsgSize, unsigned int TransferType) {
 3.   switch(ProcID){
 4.     case P_ID_Process1:
 5.      P_ID_Process1 *p1 = (P_Process1 *) ptr_Process1;
 6.      p1->Bus0busport->read(ProcID, MemFlagAddr, (void*)Flag, sizeof(FlagStruct));
 7.      break;
 8.     case P_ID_Process2:
 9.      P_Process2 *p2 = (P_Process2 *) ptr_Process2;
10.      p2->Bus0busport->read(ProcID, MemFlagAddr, (void*)Flag, sizeof(FlagStruct));
11.      break;
12.   }
13.   if (TransferType == UBC_READ){
14.       // the memory is empty
15.       if (Flag->Value < MEMORY_M0_EMPTY + MsgSize)
16.         cout<<sc_time_stamp(); printf("\tProc%d: Memory empty.\n", ProcID);
17.   } else if (TransferType == UBC_WRITE) {
18.      // the memory is full
19.       if(Flag->Value + MsgSize > MEMORY_M0_CAP)
20.         cout<<sc_time_stamp(); printf("\tProc%d: Memory full.\n", ProcID);
21.   } }
```

Figure 10. Implementation of *read_mem_flag* method (in *ubc.sc* file)

The *write_mem_flag* is invoked after the transfer has completed to update the pointers to the memory fragment. If the transfer was a data read (Figure 11. lines 3-10, for UBC_READ), the *fifoTail* will increase the address it is pointing to for the number of read bytes, and if it was a data write, the increase will be added to *fifoHead* (Figure 11. lines 11-18, for UBC_WRITE). The *fifoValue* will decrease/increase for the number of read/written bytes (lines 10 and 18).

```
 1. extern "C" void write_mem_flag(unsigned int ProcID, unsigned int MemFlagAddr,
 2.           FlagStruct *Flag, unsigned int MsgSize, unsigned int TransferType) {
 3.   if (TransferType == UBC_READ) {
 4.       // update pointers for read
 5.       Flag->fifoTail += MsgSize;
 6.       if(Flag->fifoTail >= MEMORY_M0_CAP)
 7.            // circular FIFO completed one cycle; back to init value
 8.            Flag->fifoTail = MEMORY_M0_EMPTY;
 9.       // update memory status
10.       Flag->Value -= MsgSize;
11.   } else if (TransferType == UBC_WRITE) {
12.       // update pointers for write
13.       Flag->fifoHead += MsgSize;
14.       if(Flag->fifoHead >= MEMORY_M0_CAP)
15.            // circular FIFO completed one cycle; back to init value
16.            Flag->fifoHead = MEMORY_M0_EMPTY;
17.       // update memory status
18.       Flag->Value += MsgSize;
19.   }
20.   switch(ProcID){
21.     case P_ID_Process1:
22.      P_Process1 *p1 = (P_Process1 *) ptr_Process1;
23.      p1->Bus0busport->write(ProcID, MemFlagAddr, (void*)Flag, sizeof(FlagStruct));
24.      break;
25.     case P_ID_Process2:
26.      P_Process2 *p2 = (P_Process2 *) ptr_Process2;
27.      p2->Bus0busport->write(ProcID, MemFlagAddr, (void*)Flag, sizeof(FlagStruct));
28.      break;
29.   } }
```

Figure 11. Implementation of *write_mem_flag* method implementation (in *ubc.sc* file)

```
#define MEMORY_M0_EMPTY (unsigned int) sizeof(FlagStruct)
#define MEMORY_M0_CAP   (unsigned int) 10000
#define MEMORY_M0_FULL  (unsigned int) MEMORY_M0_EMPTY+MEMORY_M0_FULL

1. class M_MEMORY_M0: public sc_module{
2.  public:
3.   SC_HAS_PROCESS(M_MEMORY_M0);
4.   M_MEMORY_M0(sc_module_name name):sc_module(name){
5.          MEMORY_M0[0] = MEMORY_M0[4] = MEMORY_M0[8] = MEMORY_M0_EMPTY;
6.          for(i = MEMORY_M0_EMPTY; i < MEMORY_M0_FULL; i++)
7.                      MEMORY_M0[i] = 0;
8.          SC_THREAD(MEMORY_M0_thread);
9.          set_stack_size(0x5000000);
10.  }
11.  unsigned char MEMORY_M0[MEMORY_M0_CAP];
12.  …
```

Figure 12. Memory Module definition (in *tlm.cpp* file)

The memory fragment is implemented as a *sc_module* with a servicing thread *MEMORY_M0_thread.* Figure 12. shows the memory implementation in the TLM definition file (tlm.ccp). It contains an array of memory locations (line 8) and a bus port (line 9) to the UBC. The communication routines that support shared memory flag set-reset are as follows:

```
1. extern "C" void send_P1_P2(void *ptr, int size){
2.    Process1 *p = (Process1 *) ptr_Process1; {
3.    FlagStruct flag;
4.      // poll flag
5.      while(1) {
6.       read_mem_flag(P1,MEMORY_M0_LOW, &flag, size, UBC_WRITE));
7.       if(flag.Value + size > MEMORY_M0_CAP)
8.                wait(10, SC_NS);  // wait 10 ns before polling again
9.        else
10.               break;
11.     }
12.     //write data
13.     p->port->write(P1, MEMORY_M0_LOW+flag.fifoHead, ptr, size);
14.     // set flag
15.     write_mem_flag(P1, MEMORY_M0_LOW, &flag, size, UBC_WRITE);
16. }
17.
18. extern "C" void recv_P2_P1(void *ptr, int size){
19.    Process2 *p = (Process2 *) ptr_Process2; {
21.    FlagStruct *flag;
22.     // poll flag
23.     while(1) {
24.        read_mem_flag(P1, MEMORY_M0_LOW, &flag, size, UBC_READ);
25.        if(flag.Value – size < MEMORY_M0_EMPTY)
26.                wait(10, SC_NS);  // wait 10 ns before polling again
27.        else
28.                break;
29.     }
30.     //read data
31.     p->port->read(P1, MEMORY_M0_LOW+flag.fifoTail, ptr, size);
32.     // set flag
33.     write_mem_flag(P1, MEMORY_M0_LOW, &flag, size, UBC_READ);
34. }
```

Figure 13. Send/Recv communication methods (in *tlm.cpp* file)

7.3. Proposed extensions to TLMest

KPN consists of a set of computing Kahn Processes and a set of Kahn Channels. TLM estimation can estimate Kahn Processes as a regular C code. However, the TLMest needs to provide time estimates that Kahn Channel needs to acquire the bus (only in inter-processor communication), synchronize each process with an intermediary buffer and transfer the data. Also, with each unsuccessful checking of the FIFO flags, the waiting period of the next transfer attempt needs to be defined. This can be done using a macro definitions stated in Figure 14.

```
#define REQ_OPB_BUS     (unsigned int)  2 // in nsec
#define ACK_OPB_BUS     (unsigned int)  1 // in nsec
#define READ_OPB_DELAY  (unsigned int)  2 // in nsec
#define WRITE_OPB_DELAY (unsigned int)  2 // in nsec
#define POLLING_DELAY   (unsigned int) 50 // in nsec
```

Figure14. Time delays definitions (in *ubc.sc* file)

Next, wait statements would be inserted in the communication code (in *ubc.sc* and *tx.sc*) to reflect the specified time delays. For example: before requesting and releasing the bus, the process would execute *wait (REQ_OPB_BUS, SC_NS)* and the arbiter would in turn execute *wait (ACK_OPB_BUS, SC_NS)* command.

8. Phase 2: Spec Model Mapping to MPSoC Platform

The second phase is already implemented in ESE and no additional changes are required. The designer defines which memory fragments created from the Kahn Channels are mapped to which Memory Module and assigns an address space to each fragment. Similarly, the fragments mapped to the TX are defined as TX FIFOs. Further, the TLMgen assigns all created Spec Channels with the same route as mapped to the same UBC and schedules them to run sequentially.
The approach is outlined in Figure 15.

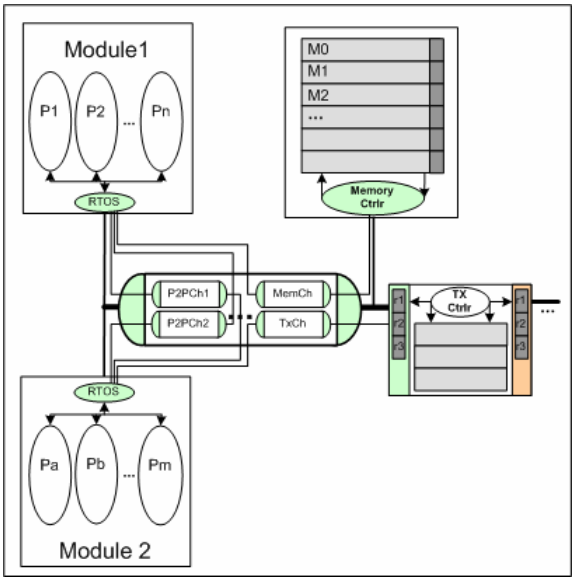


Figure 15. Spec Channel Mapping into the UBC object

Modules Module1 and Module2 contain concurrent processes P1,P2…Pn and Pa,Pb,…Pm, respectively. The processes communicate via point-to-point Spec Channels either directly (e.g. P2PCh1 and P2PCh2, from Figure 15), or indirectly (e.g. on Figure 15, channels MemCh and.TxCh, to memory and bridge, respectively.). In indirect communication, in addition to Spec Channels each pair of processes also contains a dedicated memory fragment mapped either to the memory (M0, M1, M2…) or the internal storage of the bridge (TX unit).

Since point-to-point Spec Channels execute concurrently, when mapped to the UBC object they need to be made sequential. This is done with arbitration for the access to the UBC. Each processes need to obtain permission to access its channel before using it, and release the access after the transaction is done. Therefore, all transactions within the UBC are sequential. The arbitration in UBC is implemented as a simple first come first serve mutex lock.

9. Experimental setup:

The H.264 Encoder is part of the H.264 ITU-recognized standard for compression of a stream of video data frames. The algorithm divides each data frame into sub-blocks and performs a series of transformations on each. To achieve computational speedup, the algorithm attempts to re-apply already calculated transformations to adjacent sub-blocks. This is known as intra-frame prediction. Further, it predicts similarities in adjacent data frames and re-applies performed calculations to the next frame (inter-frame prediction).

The major functional components in H.264 encoding algorithm are listed below:

- Discrete cosine transformation (DCT)
- Quantization
- Inverse quantization
- Inverse DCT
- Motion Estimation / Prediction Calculation (both for inter- and intra-frame prediction)
- Prediction Mode Selection (both for inter- and intra-frame prediction)

The simplified KPN of H.264 Encoder is shown on Figure 16. The Kahn processes are shown with 13 diagram blocks and Kahn channels are abstracted with unidirectional arrows among blocks. The majority of Kahn channels (there are 43 in total) are omitted for clarity.

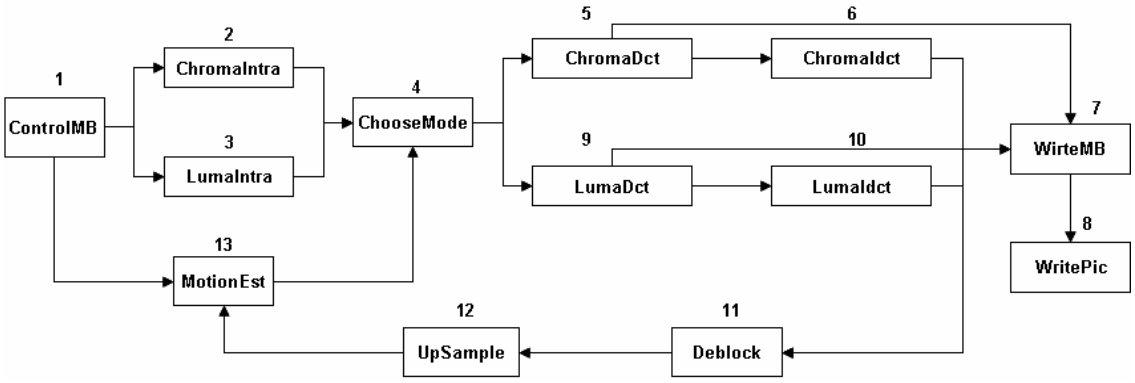


Figure 16. H.264 Application Specification

Processes *ChromaIntra* and *LumaIntra* are a part of intra-frame prediction, for both *chroma* (chromatic, colorful) and *luma* (achromatic, without color) aspect of the specter. Discrete cosine transformation and quantization are implemented in *ChromaDct* and *LumaDct*, processes (again, separately for *chroma* and *luma* aspects), while the inverse is done in *ChromaIdct* and *LumaIdct*.The heaviest computation load is encapsulated in *MotionEst* process, which performs motion estimation and prediction calculation for both intra- and inter-frame

predictions. Processes *WriteMB* and *WritePic* write the output into the output file, and *Deblock* and *UpSample* perform minor auxiliary computations, such as merging the blocks into the encoded frame (*Deblock*) and sampling them (*UpSample*).

We have mapped H.264 Encoder application on a set of heterogeneous MPSoC platforms. Following is the detailed description of the platforms.

9.1. Platform 1

The first MPSoC platform, shown in Figure 17, consists of three processors (Modules 1 to 3), two buses (UBC1 and UBC2) and a 2-port bridge (TX). Supported communication paradigms are:

- Inter-processor communication: via shared memory or TX and
- Intra-processor communication: via processor local memory.

Shared Memory (via UBC1 channel) communication is implemented between processes of Module1 and Module2. Process of Module3 supports a different communication protocol and communicates with other processes through TX and two channels (UBC1 and UBC2). Finally, multitasking and intra-processor communication is supported within Module1 and Module2 (as shown in Figure 17. with two RTOS models, highlighted green): Module1 contains 8 processes and Module2 has 4.

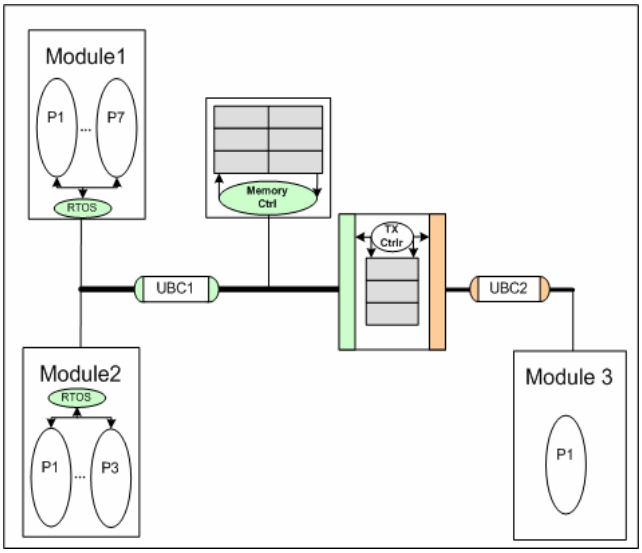


Figure 17. MPSoC Platform 1

9.2. Platform 2

The second MPSoC platform for H.264 Encoder contains five processors (Modules 1 to 5), and three buses (UBC1, UBC2 and UBC3), as shown in Figure 18. All processors are communicating through a single 3-port bridge unit (TX). Processors Module1 and Module3 are supporting multitasking and intra-processor communication (green highlighted RTOS models): Module1 contains 7 processes and Module3 has 4.

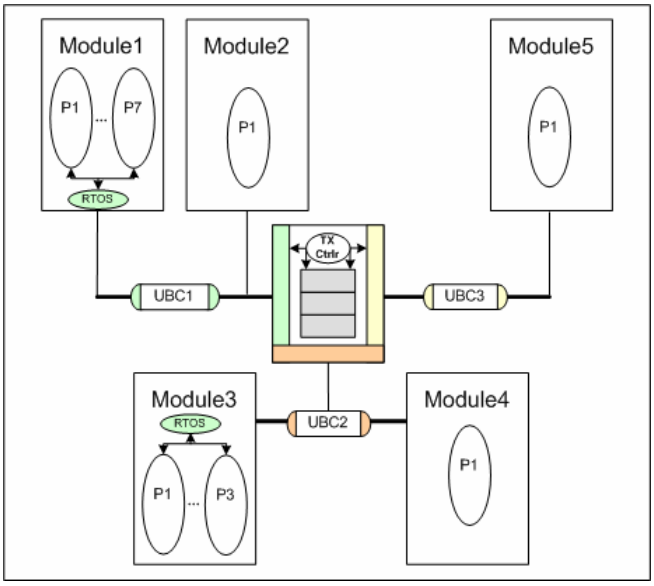


Figure 18. MPSoC Platform 2

9.3. Platform 3

The third and final MPSoC platform for H.264 Encoder has the same mapping as the platform described in the previous section (see Figure 18). However, in this platform, processes of Modules 3 and 4 are not using the bridge unit (TX) to communicate with each other, but are doing so via shared global memory.

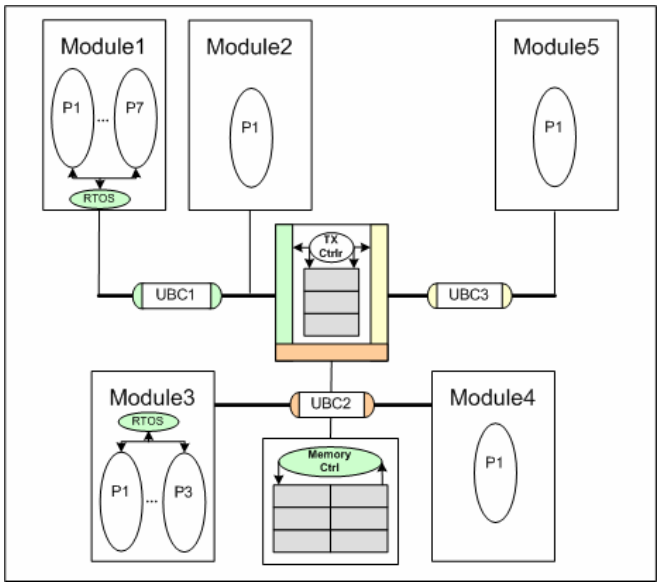


Figure 19. MPSoC Platform 3

9.4. Communication profile of H264 Encoder

The application consists of 13 processes that communicate among themselves. There are 43 communicating sender-receiver pairs of processes, exchanging the total of 998 messages per

each data frame. The average message size is 6.625 KB, however, the sizes of messages are ranging from 4 bytes to 945 KB.

The application is mapped to the described MPSoC platforms.
In Platform1 (three processors, 1st row of Table 4), only 24 pairs of processes communicate within a processors (via local memory, 2nd column on Table 4.), and 19 are accessing the bus. Further, 13 of those are using the global shared memory (3rd column, Table 4) and 6 are using a bridge TX to translate between protocols (4th column, Table 4).
In Platform2 (five processors, 2nd row of Table 4), 18 process pairs are communicating via local shared memory, and 25 are requesting the services of the bridge element (TX).
Finally, in Platform3 (five processors, 2nd row of Table 4), the KPN-to-platform mapping is the same as in Platform2. However, in Platform1, 1 pair of processes is communicating via shared memory (3rd row, 3rd column). Therefore, the traffic through the TX unit in this platform is alleviated by one (from 25 pairs to 24 pairs, 2nd and 3rd row, column 4), as compared to Platform2 TX traffic.

	Intra-processor comm. pairs	**Inter-processor comm. pairs**	
Name:	**Local Sh. Mem.**	**Global Sh. Mem.**	**TX**
Platform1	24 (55.82%)	13 (30.23%)	6 (13.95%)
Platform2	18 (41.86%)	0	25 (58.14%)
Platform3	18 (41.86%)	1 (2.32%)	24 (55.81%)

Table 4. Communication profile of the H264 Encoder mapped to Platform1 to Platform3

9.5. Results:

The created TLMs of 3 platforms can be separated into two types:

- purely functional models with no timing annotations, or **untimed TLMs**, and
- TLM with estimated timing for both communication and computation, or **timed TLMs**

For models of both groups, we measured the execution time of their simulations on a Pentium with 4 CPUs, running on 3 GHz and with 1 GB of RAM. The results of these measurements are shown on Table 5, columns 3 and 4, for untimed and timed TLMs, respectively.
In addition, the timed models estimate how long the implementation of that particular system on FBGA board[1] will take to run. The results of time estimation of timed TLMs for all 3 platforms are shown in column 5 of Table 5.

Name:	**Platform Description:**	**Untimed TLM**	**Estimated time TLM**	
		Simulation exe. time	Simulation exe. time	**Simulated time (estimate)[1]**
Platform1	3 modules, 2 buses, 1 2-port TX	47.88 s	TBD	TBD
Platform2	5 modules, 3 buses, 1 3-port TX	54.51 s	635.36 s	824.60 s
Platform3	5 modules, 3 buses, 1 3-port TX	53.96 s	640.95 s	814.58 s

Table 5. Simulation time of TLM models of the H264 Encoder mapped to Platform1 to Platform3

The simulation runs the fastest for the untimed TLM (columns 3 and 4 of Table 5) of Platform1, because more data transfers are handled within a processor and with fast accesses to its local memory (24 P2P channels of Platform1 versus 18 in Platform2). As the platform complexity increases, the number of bus accesses in inter-processor transactions grows. Since bus transactions need to be serialized, they will take more time to complete. Finally, the bridge unit

[1] Modules mapped to SW PEs are implemented as *Microblaze* soft-core processors; the buses are OPT on-chip buses; the TX units are implemented as HW IP units.

adds additional delays since the processes must wait for the TX to be ready and have enough buffer space available to perform the requested service.

10. Conclusion:

We presented simple and automatic 2–phase method of converting KPN application into a TL model that reflects the MPSoC platform. In the first phase, the KP channel is converted into a memory fragment accessible with to point-to-point blocking channels. The designer maps the fragment to (a) a global memory unit, (b) local memory of the processor or to (c) a buffer of a bridge unit and the channels are automatically assigned to appropriate bus models. The second phase consists of serializing the channel accesses within each bus model in the platform.

Simulation results of the TLMs generated with our method demonstrates the benefits of our approach. With a small increase in simulation time of multi-core platforms (as compared to a single-core reference platform), it provides the user with a fast functional validation of the system that accurately reflects the underlying MPSoC platform.

For future work, our research is focused on generating TLMs with timing annotations for system computation and communication. We are also working on extending the supported inputs to communication paradigms with mailboxes and semaphores. Finally, our future work will include support for platform templates of more complex MPSoC and NoC architectures.

References:

[1] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, E. Deprettere, ***"System Design using Kahn Process Networks: The Compaan/Laura Approach"***, In Proceedings of "7th International Conference Design, Automation and Test in Europe (DATE'04)", pp. 340-345, Paris, France, Feb. 16-20, 2004.

[2] L. Yu, S. Abdi, D. Gajski: "**Transaction Level Platform Modeling in SystemC for Multi-Processor Designs"**, Technical report TR07-01, UC Irvine, 2007

[3] M.J. Rutten et al. ***"Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing"***, Workshop on Parallel and Distributed Computing in Image Processing, Video Processing and Multimedia (PDVIM'02), Fort Lauderdale, Florida, USA, April 15, 2002.

[4] G. Kahn, "***The Semantics of a Simple Language for Parallel Programming***", Proceedings of Information Processing '74, pp. 471-475, Stockholm, Sweden , August 5-10, 1974.

[5] G. Kahn and D.B. MacQueen, "***Coroutines and Networks of Parallel Programming***", Proceedings of Information Processing '77, pp. 993-998, August, 1977.

[6] Sangiovanni-Vincentelli et al. "***A next-generation design framework for platform based design***." In Conference on Using Hardware Design and Veri_cation Languages (DVCon), February 2007.

[7] H. Kopetz, R. Obermaisser, C.E. Salloum, and B. Huber. ***"Automotive software development for a multi-core system-on-a-chip".*** In Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems (SEAS'07), Washington, DC, USA, 2007.

[8] Hermann Kopetz and Gather Bauer. "***The Time-Triggered Architecture"***. Proceedings of the IEEE, 91(1):126{113, January 2003.

[9] Gunar Schirner, Andreas Gerstlauer, and Rainer Doemer***." Abstract Multifaceted Modeling of Embedded Processors for System-Level Design"***. In Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC), Yokohama, Japan, January 2007.

[10] Thorsten Groetker, Stan Liao, Grant Martin, and Stuart Swan. ***"System Design with SystemC".*** Kluwer Academic Publishers, 2002.