Automatic Generation of Transducer Models for Bus-based MPSoC Design

Hansu Cho, *Member, IEEE*, Lochi Yu, *Member, IEEE* and Samar Abdi, *Member, IEEE*

Abstract— This paper presents methods for automatic generation of models of Transducer, a highly flexible communication module for interfacing Multi-Processor System-on-Chip (MPSoC) components. We describe the transducer architecture, comprising the bus interface, high-level communication controllers and buffer management blocks. The well defined architecture of the transducer enables automatic generation of its Transaction-level and Register-transfer level (RTL) models. Moreover, the simple interface of the transducer provides for a well defined software interface, making it easy to update the software after changes in MPSoC platform. Our experimental results show that MPSoC design for industrial-size applications, such as MP3 decoder and JPEG encoder, greatly benefits from automatic generation of transducer models. We found productivity gains of 9-23X due to significant savings in modeling effort. On the quality axis, we show that MPSoC communication design using automatically generated transducers has very little overhead in communication delay over a fully-connected point-to-point communication architecture. Finally, we show that our automatically generated TLMs greatly reduce the system-level modeling time and provide a fast executable model for early functional validation.

Index Terms— System level modeling, Multi-processor System-on-Chip design, Communication architecture

1 Introduction

THE demand for higher computational power in contemporary embedded applications has led to increasing adoption of Multi-processor System-on-Chip (MPSoC) platforms. However, the modeling and design of the MPSoC system, in particular the communication architecture, can be complex, time consuming and error prone. In particular, the development of cycle-accurate models for MPSoC design and validation can cause significant delays in the project. Therefore, there is a need for new automatic model generation methods and tools at the system level that can alleviate the problem of modeling and design of MPSoCs.

The MPSoC communication design problem has been extensively studied in the SoC design community. There are two major on-chip communication architecture templates in use today: bus-based, and Network-on-Chip (NoC) [1]. NoCs are advantageous for high-throughput, massively parallel applications, but may be an overkill for embedded applications such as image processing, audio and video, where the use of a bus-based system with a few processors cores and hardware accelerator cores may suffice. Such platforms may require multiple buses to minimize arbitration delays or to integrate IP cores with different interface protocols, as in the case of heterogeneous MPSoC systems. In this paper, we present Transducer, a highly flexible and scalable communication module for enabling system level communication in bus-based MPSoC platforms. Well defined transducer architecture enables automatic generation of its synthesizable models from a system-level specification.

Automatic transducer model generation enables a true system-level communication design solution. An application model with concurrent tasks, communicating over abstract point-to-point channels can be mapped to any bus-based MPSoC platform. The transducer logic that implements the system level communication over a single or multiple buses, can then be automatically generated. The contributions of our work, described in this paper, are (i) definition of a flexible and scalable transducer architecture that can be adapted to any bus configuration in a MPSoC platform; (ii) methods for automatic generation of synthesizable RTL and SystemC transaction-level models of the transducer; and (iii) quantification of the productivity and design quality metrics for MPSoC design with automatically generated transducer models.

2 Related Work

We target automatic model generation for functional validation, performance analysis and implementation of bus-based communication in MPSoCs. The related research topics can be broadly categorized into automatic MPSoC model generation and MPSoC communication synthesis.

The SystemC language [2] has been a key driving force behind the industrial adoption of higher level modeling above the RTL abstraction. In particular, various abstractions, based on modeling detail and use cases, have been proposed for Transaction Level Models (TLMs) of MPSoCs [3, 4, 5]. Gerstlauer et al. have proposed automatic model generation using well defined model semantics for MPSoC platforms [6]. Their approach is to create a transaction level virtual prototype of the system that includes models of the processor cores, embedded software

- *Hansu Cho is with the Design Solution Lab, DMC R&D Center, Samsung Electronics, Suwon-City, Korea. Email: hansu.cho@samsung.com.*
- *Lochi Yu is with the Department of Electrical and Computer Engineering, University of Costa Rica.. E-mail: lochiyu@eie.ucr.ac.cr.*
- *Samar Abdi is with the Department of Electrical and Computer Eng., Concordia University, Montreal, Canada. Email samar@ece.concordia.ca*

Manuscript received (December 12, 2011).

stack and the communication architecture. Although they consider transducer based communication architectures, the semantics and internal structure of the transducer is not discussed. Also, there is not path to eventual RTL implementation of the transducer. Cornet et al. propose a methodology for introducing timing into TLMs by successive refinement, but do not consider platforms with multiple PEs and multi-bus platforms [7]. Chen et al. propose automatic generation of TLMs, at different abstraction levels, from a formal model of the MPSoC design [8]. However, they consider only pairs of master-slave components and do not support shared buses. Van Moll et al. propose a methodology to create protocol-specific bus-cycle accurate interfaces and transactors [9]. Their objective is to comply with the TLM 2.0 bus modeling standard. However, they do not support modeling of MPSoC communication IPs such as bridges and transducers.

MPSoC communication synthesis is also an active area of research. Bombieri et al. have proposed a technique for deriving RTL implementation of IP interfaces from Extended Finite State Machines (EFSMs) and applied it to synthesis from TLM to RTL [10]. However, the synthesis applies to protocols at the bus-word level and not end-to-end transactions across an MPSoC platform. Thompson et al. have proposed the Daedalus framework for exploration and prototyping of MPSoCs [11]. However, their communication architecture is a fixed cross-bar switch, which does not provide the flexibility for true heterogeneous MPSoC design. Gladigau et al. have proposed an MPSoC synthesis methodology based on formal specification, but their platforms only have 2-3 PEs [12]. The communication is performed using queues implemented in shared memories, which is not scalable to larger MPSoCs and does not address heterogeneous buses.

Grasset et al. have proposed automatic generation of RTL wrappers for IPs in an MPSoC [13]. However, the wrappers are manually composed to provide network-level services over the IP's bus interface and do not take application-specific communication needs into account. As such, they operate at a lower level of abstraction than the transducer and do not support model automation. Zimmerman et al. have proposed generation of protocol adapters to integrate synthesized IPs into MPSoCs [14]. However, their technique relies on matching attributes of specific bus protocols to those of a generic bus. In general such matching may not always be feasible. Moreover, the abstraction of the adapters is not shown to be applicable to hard IPs in an MPSoC platform. Watanabe et al. have proposed a protocol transducer that acts as a bridge between IPs with different interface protocols [15]. It can translate only non-blocking and out-of-order protocols, so is limited in scope. The technique has also not been demonstrated to scale to heterogeneous MPSoCs.

With regard to the transducer architecture itself, the most obvious comparison point is the on-chip router in NoCs [1]. However, for most embedded applications that are not massively parallel, a bus-based MPSoC platform with heterogeneous cores may be sufficient to meet the application's performance needs. Since the application task structure and inter-task communication requirements are known a priori, we do not need the complex dynamic routing mechanism and routing tables of an on-chip router. Instead, the transducer implements a static store-forward transaction mechanism for routing the packets from one processing element to another.

Our automatic transducer model generation techniques are distinct in that they target TLMs for functional validation and FPGA prototypes for performance validation and implementation. Therefore, the transducer models are defined in both synthesizable RTL and SystemC TLM. In our experience, TLMs are easier to build and are useful for early functional validation of the application. Othe other hand, FPGA prototypes execute at close to real-time execution speed and support cycle level execution of the system, which makes them ideal for accurate performance validation and implementation.

3 System Level Design Methodology

Figure 1 shows the role of system-level modeling in a platform based design methodology [16]. We start with a system level specification as shown on the LHS. The specification consists of the application, the MPSoC platform and a mapping from the application objects to platform objects.

The application model is an executable, platform-independent specification of the system's functionality. This model consists of concurrent tasks (*P1, P2, P3*), communicating using blocking First-in-First-Out (FIFO) channels (*C1, C2*). Tasks are symbolic representations of functions specified in C or C++. All channels are point-to-point and lossless. They are implemented using the built-in FIFO channel (*sc_fifo*) available in the SystemC library. These modeling constructs allow us to create models for a wide variety of streaming applications. The application model can be executed independently for functional validation of the application. Application model simulation can help uncover logical bugs in the application description such as deadlocks or bottlenecks in the FIFO channel. However, such simulation will not validate the platform specific implementation details of the design.

A typical MPSoC platform consists of Processing Elements (PEs), such as CPU cores (*CPU 1* and *CPU 2*), hardware and IP blocks (*IP*). The communication architecture of the MPSoC platform may consist of several buses (*Bus1* and *Bus2*). Multiple buses may be connected using a transducer module. For instance, transducer Tx2 connects Bus1 and Bus2 to enable communication between CPU1 and IP or between CPU2 and IP. If multiple masters are connected to the same bus (CPU1 and CPU2 to Bus1), a single port transducer (*Tx1*) can be used to communicate between them.

A mapping from the application to the platform is defined as illustrated in Figure 1. Tasks in the application model are mapped to PEs. Channels are mapped to routes that may go through transducers. For instance, channel *C1* is mapped to the route *CPU1*→*Bus1*→*Tx1*→*Bus1*→*CPU2*. Similarly, channel *C2* is mapped to *CPU1*→*Bus1*→*Tx2*→*Bus2*→*IP*. For the purposes of our design specification, we also incorporate the task information in the route

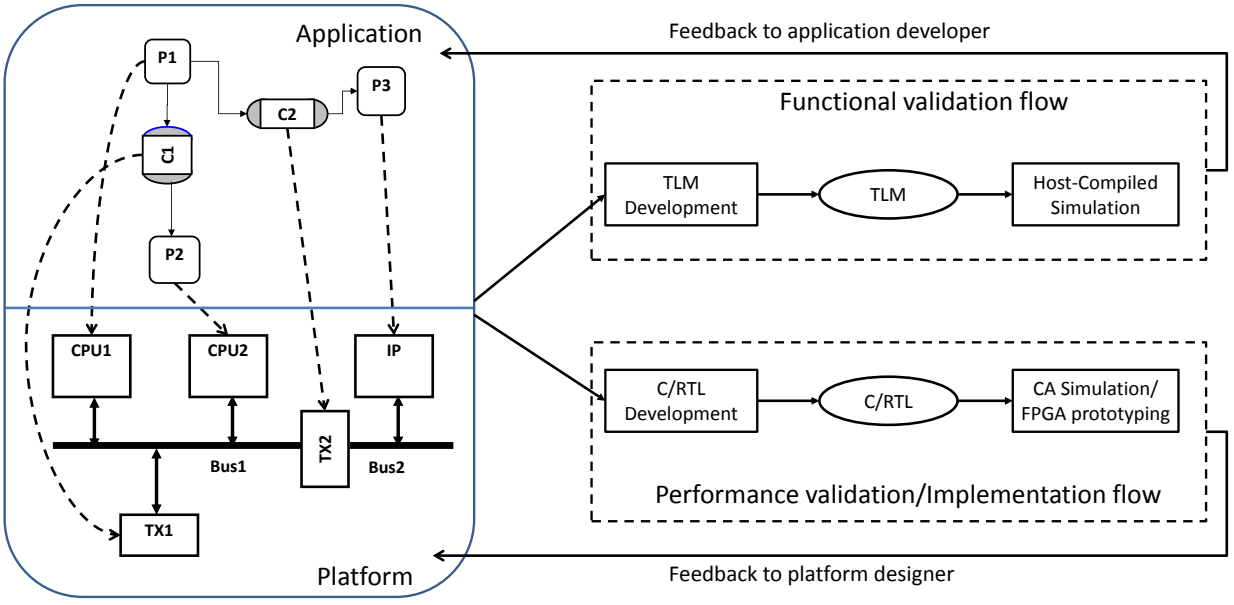

Fig. 1. Platform-based system level design methodology. The input to both TLM and RTL model development is the application mapping on the platform. TLMs are used for functional validation and application development, while the synthesizable RTL models are used to evaluate performance and serve as input to design implementation.

definition. Since we allow multiple application tasks to be mapped to the same PE, we need to disambiguate between routes of channels between different task pairs but between same PE pairs. As such, the complete specification of the route for channel C1 would be *CPU1(P1)→Bus1→ Tx2→ Bus2→CPU2(P2)*. The mapping of the channels determines the implementation of the transducers, as we shall see in Sections 4 and 5. The mapping serves as an input to both the functional validation and performance validation/implementation flows as shown in Figure 1.

The mapping is used to create an abstract TLM of the design. The TLM is typically modeled using a system level design language such as SystemC. As such, the TLM can be compiled and executed on the host machine, thereby providing very high speed validation. Therefore the TLM is ideal for early pre-silicon validation, before the hardware has been delivered. It can be used by software developers to validate and debug their applications. However, designers would need to develop the SystemC TLM manually. If the TLM semantics are well defined, we can alleviate this problem by automatically generating the TLMs as explained in Sections 9 and 10.

Although TLMs at several abstraction levels have been proposed by the Open SystemC Initiative (OSCI), none of them can be considered accurate enough for accurate performance validation. For accurate feedback of performance metrics, the MPSoC platform needs to be modeled at the cycle accurate (CA) level, which can slow down simulation by several orders of magnitude. An alternative is to prototype the MPSoC platform on FPGA. The mapped application can then be executed at close to real-time speeds on the prototyping board. However FPGA implementation would require synthesizable RTL models to be developed for all the components in the MPSoC platform. For most components, such as processors, IPs and bus controllers, such RTL is usually available off the shelf. However, the hardware and software needed to implement the abstract channel based communication in the application is developed for the specific design. By clearly defining the structure and semantics of a highly flexible communication module, such as the transducer, we can automatically generate the communication logic, as described in Sections 5, 6, and 7.

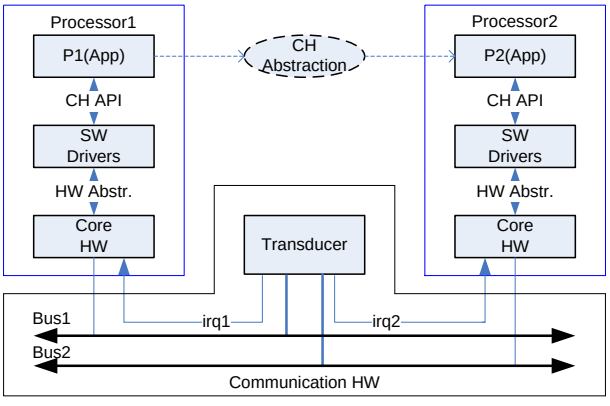

Fig. 2. System-level MPSoC communication design.

4 Transducer-based MPSoC Communication

Figure 2 shows a simple example of our MPSoC communication model to illustrate our design methodology. The executable application model is a set of concurrent tasks (*P1* and *P2*) communicating over abstract channels (*CH*). For the class of streaming applications that we are targeting, we consider only blocking first-in-first-out (FIFO) channels. The channel data type is considered to be raw bytes. These considerations are put in place to simplify the design of the transducer hardware and they do not in any way restrict the communication models used at the application level. Channels with complex data types or

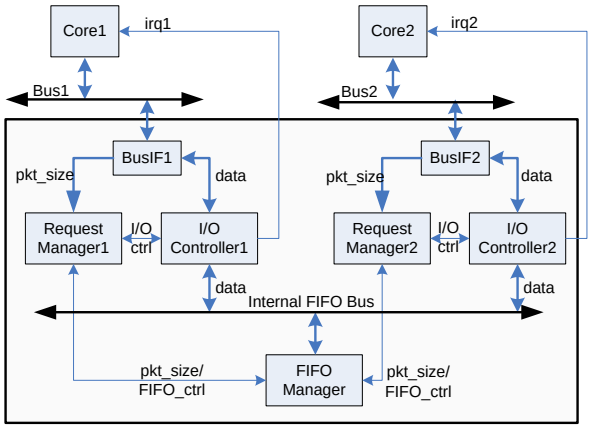

Fig. 3. Transducer architecture. The key components are the bus interface, request manager, I/O controller and the FIFO manager.

other communication mechanisms, such as non-blocking FIFO, handshake, priority queues and so on, can easily be built on top of blocking FIFOs in a layered fashion [6]. The higher layers implementing the complex channels on top of type-less blocking FIFOs are realized in software.

The application model is not cognizant of the platform and can be executed independently for application validation. However, it is not useful for validating the implementation. During system-level design, the tasks are mapped to two different PEs (Processor 1 and 2), which are connected to two different buses (Bus 1 and 2, respectively). The channel *CH* is implemented on this platform.

A transducer is introduced to support the end-to-end transaction from P1 to P2 that was originally supported by *CH* in the abstract application model. The interrupt signals (irq 1 and 2) are added to provide low-level synchronization between the transducer and the PEs. The software drivers on the PEs break down the abstract data into raw data packets and implement the CH semantics on top of the communication semantics of the transducer. As such, the software drivers provide the same channel API as the one provided by CH. Therefore, the tasks from the application model can be reused as is in the implementation. The implementation model consists of the RTL description of the hardware platform (including transducer) and the C/C++ description of the application tasks and the software drivers. This model can be used by embedded development tools, such as the Xilinx EDK [17], for implementation and validation on board.

5 Transducer Architecture

Figure 3 shows the top level internal blocks of the example transducer in Figure 1 [18]. The transducer consists of four types of components: Bus Interface (BusIF), Request Manager, I/O Controller, and FIFO Manager. For each bus that the transducer connects to, a unique set of bus interface, request manager and I/O controller is instantiated. For each bus, the transducer is assigned a set of addresses equal to the number of distinct routes in the platform that include the bus. These addresses, which we will refer to as being in the request manager range, are used for making transaction requests to the transducer as we

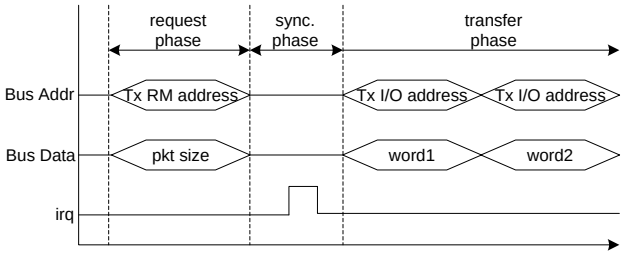

Fig. 4. High-level transducer protocol.

will explain below. A separate address (Tx I/O address) is assigned for reading (writing) the packet data from (to) the transducer. A FIFO manager and an internal FIFO bus is instantiated inside each transducer. All the I/O controllers connect to the shared FIFO bus. Each of the request managers has a dedicated interface to the FIFO manager.

The above architectural choices enable the transducer to support large heterogeneous MPSoC platforms, thereby enabling design scalability. The simple template of the transducer allows multiple sets of Bus I/F, Request Manager and I/O controller to be instantiated, thereby supporting as many buses as needed. Decoupling the bus interfaces from the shared FIFO data storage, using Request Manager and I/O controller, allows buses with different clock speeds and protocols to be integrated in the platform. Finally, the shared FIFO and the single internal FIFO bus help us avoid the wire density issues associated with point-to-point communication architectures. If the shared FIFO becomes a bottleneck, a separate transducer may be instantiated, and some of the channels routed through it, to distribute the communication load.

Figure 4 shows the high-level protocol of the transducer. Recall that the transducer implements blocking FIFO semantics to emulate the abstract channels. The transaction between a PE and transducer is divided into three phases. In the first phase, the application on the PE makes a transaction request to the transducer. The bus address used for the request (TxRMAddress) encodes the route information and, consequently, the type (read or write) of the transaction. The data is the size of the packet being read or written (pkt size). The request phase is followed by the synchronization phase, in which the transducer checks if there is enough available space/data in the internal FIFO for the requested transaction. The synchronization phase can be arbitrarily long depending on the FIFO status and competing requests by other PEs on the bus. If the transducer determines that the transaction can be completed, it sends an interrupt to the PE to initiate the final phase of data transfer, in which the packet data is read (written) from (to) the internal FIFO.

5.1 Bus Interface

The bus interface implements the slave side of a given system bus protocol and includes additional logic to send the data either to the request manager or I/O controller based on the address. Figure 5 shows its behavior with an abstract Finite State Machine (FSM). In state S0, it is polling the address bus. If the bus address belongs to request manager range, indicating a request phase, it goes to RM1

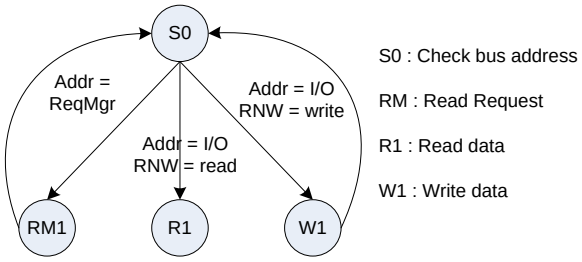

Fig. 5. FSM for Transducer Bus Interface.

state, where it reads the requested packet size from the bus and stores it in the corresponding internal register for processing by the request manager. If the bus address is the I/O register address it goes to state R1 for data read or W1 for data write depending on the transaction type.

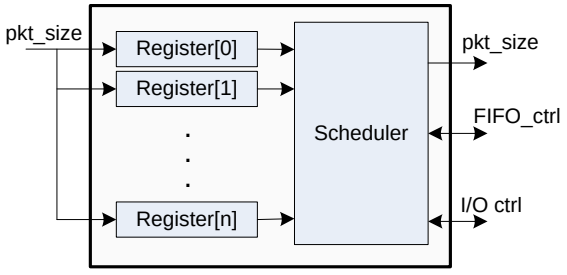

Fig. 6. Structure of the Request Manager.

5.2 Request Manager

As explained previously, and illustrated in Figure 6, there are multiple registers in the request manager, one for each of the possible routes going through the transducer. A simple first-come-first-serve (FCFS) or round-robin (RR) scheduler is implemented to prioritize requests.

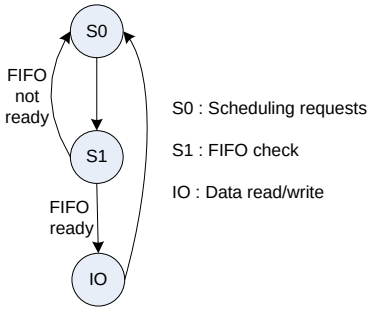

Fig. 7. Request Manager FSM.

Figure 7 shows the behavior of the request manager as an FSM. In S0 state, it selects one request, based on scheduling policy, and goes to state S1. In state S1, it checks the FIFO status. For a write transaction, the FIFO must be able to fit the packet in its buffer. For a read transaction, there must be enough data in the buffer to proceed. The FIFO_ctrl and pkt_size signals are used to detect the readiness of the FIFO. The route index (index of the register corresponding to the transaction route) is sent to the FIFO along with the packet size. If the FIFO is ready, the request manger goes to state IO. Otherwise, it returns to S0 to select a new request. In state IO, the request manager clears the request being serviced and instructs the I/O controller to send an interrupt to the processor, using the I/O_ctrl signal. The request manager stays in state IO until the FIFO reads/writes all packet data through the IO controller. Upon completion, the FIFO issues the *done* signal, returning the request manager to state S0.

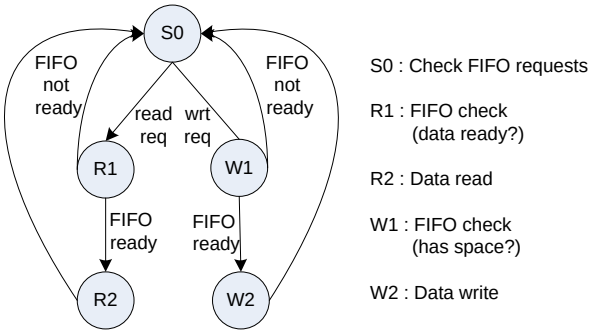

Fig. 8. FIFO Manager FSM.

5.3 I/O Controller

The I/O controller has two functions. First, it is used for packet-level synchronization with the PEs. The IO_ctrl signal from the request manager indicates that the FIFO is ready for data transfer. The I/O controller sends an interrupt to the PE to indicate the completion of the synchronization phase and the start of data transfer. Second, the I/O controller performs bus word translation between the PE bus and the FIFO bus. The transducer can be connected to multiple buses, each with different data width. The bus words are therefore reformatted by the I/O controller to match the FIFO bus width.

5.4 FIFO Controller

Figure 8 shows the FSM for the FIFO manager. It has its own scheduler and partitioned memory. The FIFO is partitioned into circular buffers, each corresponding to a unique route in the platform that includes the transducer. The size of the partitions is specified by the designer and must be at least as large as the largest packet sent on the route. Alternately, the software drivers must ensure that the packet size for an end-to-end transaction does not exceed the smallest partition for that route in the transducers.

In state S0, the FIFO scheduler selects the request from the request manager to service next. Once a FIFO request is selected, it goes to state R1 or W1 depending on the request type. In R1 or W1 state, the FIFO manager checks the appropriate partition to determine if there is enough space for a packet write or enough data for a packet read. A FIFO_ready signal from the FIFO notifies the request manager if the FIFO is ready. The FIFO manager also sends a separate ack signal to the request manager to notify that the request has been processed. This is done because the time it takes to process a FIFO request can be non-deterministic. Furthermore, the FIFO and the request manager may run on separate clocks. Therefore a handshake communication between the request manager and FIFO is needed. If the FIFO is ready, the data transfer is done in states R2 or W2 else, the FIFO manager returns to state S0 to select a new request.

Listing 1. Software interface for sending data from P1 to P2

```
1: Send_P1_P2 (void *data, uint pkt_size, uint route_id) {
2:    uint TxID = GetFirstTX (route_id);
3:    uint TxRMAddr = GetRMSendAddr (TxID,
                              route_id);
4:    uint TxIOAddr = GetIOAddr(TxID);
5:    write (TxRMAddr, pkt_size);
6:    wait_for_interrupt (TxID);
7:    for (i = 0; i < pkt_size; i++)
8:          write (TxIOAddr, data + i*WordSize);
9: }// end Send_P1_P2
```

6 Software Interface

The well defined architecture and system level protocol of the transducer enables us to precisely define the software driver functions for end-to-end communication. Listing 1 shows pseudo code of an example driver function to send data from PE P1 to P2 using the route encoded in route_id. The packet size for the transactions is defined in the variable pkt_size. We start by identifying the transducer (TxID) to which the data must be written, using a simple table lookup function, GetFirstTx (line 2). Next, we determine the address, TxRMAddr, for the register that keeps requests for sending data from P1 to P2 via the given route (line 3). We also determine the transducer IO address using a lookup function GetIOAddr (line 4). The send request is then made by writing the packet size into the request register (line 5). The processor then waits on an interrupt from the specific transducer, signifying the request has been granted (line 6). Finally, the packet data is written to the transducer's internal FIFO to complete the sender's end of the transaction (lines 7-8).

The receiver end of the driver is defined similarly. A layer based approach can be used to automatically generate the software drivers at the system level as we have previously discussed in [19]. The software layers implement the abstract end-to-end communication, specified in the system level application model, using lower level drivers similar to the one shown in Listing 1.

7 Automatic Transducer RTL Generation

The well defined architecture of the transducer enables us to automatically generate synthesizable transducer models from a system level specification. The transducer parameters and the platform configuration defined in the specification can be used to generate and instantiate the transducer's subcomponents described in Section 5.

7.1 Transducer Interface Generation

Listing 2 presents the pseudo-code for generating the interface components of the transducer, tx, namely the request manager (rm), bus interface (bif), and I/O controller (ioc). We are given the set of buses (buses), the set of PEs (Processors), and the set of routes in the system (routes). We iterate over all the buses in the system that are connected to the transducer tx (line 1). We initialize two variables: i is used as an index for the request registers in the request manager and addr is used as an address variable to hold the request addresses. The variable addr is initialized to be the lowest address in the request manager range (rm_range) defined in the system specification (line 2). We first generate part of the bus interface module (bif) by instantiating the slave logic for the bus type of b (line 3). We then create the I/O controller (ioc) with the basic functionality of data width matching between b and the internal FIFO bus (line 4).

Listing 2. Pseudo-code for generation of Transducer-bus interface

```
1: for all b ∈ Buses, s.t. tx connects to b, do
2:       i = 0; addr = min (rm_range(b, tx));
3:       bif = generate_bus_interface (b, tx);
4:       ioc = generate_io_controller (b, tx);
5:       for all p ∈ Processors, s.t. p connects to b, do
6:          for all r ∈ Routes, s.t. p ∈ r, do
7:                   rr_i = generate_register (wordsize(b));
8:                   bif →assign_address (addr, rr_i);
9:                   ioc →add_interrupt (i, p);
10:                  i++; addr+=wordsize(b);
11:         end for
12:         bif →assign_address (addr, ioc);
13:         rm = generate_request_manager(tx, b,
                                    RM_POLICY, rr)
14:   end for
15:    txif = generate_tx_if (tx, b, bif, ioc, rm);
16: end for
```

Under the top level bus loop, we iterate over all the processors (p) connected to the bus (line 5). For all the routes in the platform that either begin or terminate at p (line 6), we insert a request register (rr_i) of the data word size of bus b (line 7). The logic to enable the writing of rri for bus address addr on b, is generated inside the bus interface block (line 8). Also, interrupt generation logic is added to the I/O controller for sending interrupt to processor p when the request made in register rri is accepted by the request manager (line 9). Once the request registers have been added, we generate logic in the bus interface to send (receive) data to (from) I/O controller on address addr on bus b (line 12). Finally, the request manager is generated for the given transducer tx, bus b, the specified request scheduling policy (FCFS or round-robin) and the set of request registers (line 13). After exiting the iterations over the PEs and routes, the top level transducer interface (txif) is instantiated in tx. The top level connections between the request manager and the FIFO of *tx*, as well as those between the I/O controller and the FIFO bus are made at this time.

7.2 FIFO Manager Generation

The pseudo-code for generating the FIFO manager is presented in Listing 3. A partial FIFO manager module is created by instantiating a memory buffer of size FIFO_SIZE and a scheduler with FCFS or round-robin policy, as defined in the system level specification (line 1). The FIFO_SIZE is obtained from the specification as well, by adding the partition size (PSIZE) of each route that

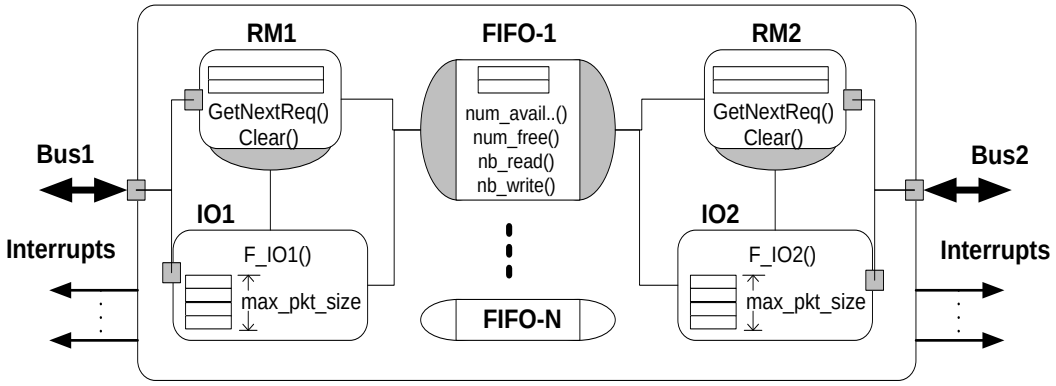

Fig. 9. TLM structure of transducer.

goes through tx. Recall that the partition for a route must be at least as large as the maximum packet size for all end-to-end application channels mapped to that route. Finally, an interface to the internal FIFO bus is created and the port connections are made.

Listing 3. Pseudo-code for generation of FIFO manager

```
1: fifo = generate_fifo (tx, FIFO_SIZE, FIFO_POLICY);
2: for all r ∈ Routes, s.t. tx ∈ r, do
3:         head = generate_head_register
                    (tx→MAX_FIFO_ADDR_SIZE);
4:         tail = generate_tail_register
                    (tx→MAX_FIFO_ADDR_SIZE);
5:         fifo→generate_partition (head, tail, PSIZE[r]);
6:         rm = get_request_manager(tx, r);
7:         fifo→ generate_ready_logic (rm,
                    get_route_index (rm, r), head, tail);
8: end for
```

To generate the partitions in the FIFO and the internal logic of the FIFO manager, we iterate over all the routes, r, that go through the transducer (line 2). A partition with a circular buffer is created per route as explained in Section 5.4. Two registers: head and tail are generated to manage the read and write operations into the circular buffer (lines 3-4). The partition is then created in the buffer memory and the read/write logic is implemented (line 5). The interface to the request manager is created by first selecting the appropriate request manager (rm) in tx, corresponding to the route r (line 6). As explained in Section 5.2, the request manager checks the readiness of the FIFO for a given transaction by sending the index of the route. Therefore, we generate the logic for the FIFO to check its readiness by correlating the route index of r with the head and tail registers (line 7). The logic compares the corresponding head and tail register values to determine the availability of space/data in the partition corresponding to the route index. The top level of the transducer is created simply by hierarchically composing the FIFO, FIFO bus, request managers, bus interfaces and I/O controllers.

8 Transducer TLM Semantics

The purpose of the transducer TLM is to integrate the model into a larger functional TLM of the system. Our goal is to define a purely functional untimed model, while explicitly modeling the abstract transducer structure and maintaining the causal order of transactions specified in the application.

The top level structure of the transducer is shown in Figure 9. The transducer model interfaces to the rest of the system model using well defined ports to bus channels. It synchronizes with the software interface methods using global interrupt events, each corresponding to a unique interrupt signal in hardware. The bus channel, that we will refer to as *Universal Bus Channel (UBC)*, is a generic abstraction of addressable shared bus. The transducer module itself consists of instantiations of bus interfaces connected to UBCs, request managers, IO controllers and FIFOs. A single mutex is defined to protect access to all FIFOs, thereby emulating the single shared FIFO bus of the transducer as shown in Figure 3. An event, *FifoTrEv*, also defined in the transducer module scope, is notified each time a FIFO is read or written.

The number of FIFOs is equal to the number of unique routes going through the transducer. We also generate a set of route identifiers for each transducer-bus connection. Therefore, for a given interface between transducer *Tx* and bus B, we create unique route IDs for all routes of the type *…→B→Tx→… or PE(P)→Tx→B→…* These route identifiers are simply consequetive positive integers starting from 0. A route id is used to index into the request buffer memory, FIFOs and interrupts as we will see later.

8.1 Universal Bus Channel (UBC)

The UBC implements two interfaces: *master* and *slave*. The master interface defines virtual functions for bus *write* and bus *read*. The slave interface defines a virtual function, *MemServe*, which enables masters to access the addressable memory of the slave components connected to the same bus channel. A mutex (*sc_mutex*) is used to model an arbiter. *Address* and *Data* variables are defined as per the address size and word size specification of the bus. A Boolean variable *RNW* (*Read Not Write*) is used to indicate the type of transaction. An event, *AddressSet*, is

Listing 4. UBC write method

```
1: UBC::write (uint Addr, void * WrData, uint TrSize) {
2:    Arbiter.lock(); // get bus
3:    RNW = false;
4:    Data = WrData;
5:    Size = TrSize;
6:    BusAddr = Addr;
7:    notifyall (AddrSet);
8:    wait (SlaveDone);
9:    Arbiter.unlock(); // release bus
10:} // end bus.write
```

used to notify the slaves that a new address has been put on the bus by a master. A corresponding event, *SlaveDone*, is used by the slave to notify the master that the read or write transaction is complete. PEs connect to the UBC as masters and transducers connect as slaves.

Listing 4 shows the pseudo-code for the UBC write method. The UBC implements the *write* method by locking the mutex, indicating arbitration. Once the mutex is locked, the master proceeds by setting the RNW field to false (indicating a write), copying the pointer to passed data (*WrData*) into the channel's *Data* variable, the transaction size into the channel's *Size* variable and the given address location into the channel's *Address* variable. Then, it notifies the *AddressSet* event to wake up the *MemServe* method of the slaves. It then waits for the *SlaveDone* event to be notified. The slaves check if the address is in their range. The target slave copies the data from the channel's data variable into its appropriate memory location and notifies the *SlaveDone* event. The master is woken up and releases the bus by unlocking the *Arbiter* mutex. The UBC read function is implemented similarly, except that the data is copied over by the master *after* the *SlaveDone* event has been notified.

The UBC implements the *MemServe* function of the slave interface as shown in Listing 5. Upon calling *MemServe*, the slave thread waits on the *AddrSet* event. Once the event is notified, if the address given by the master falls in the slave's address range [*LOW, HIGH*], the slave checks for the RNW flag to determine the transaction type. If it is a write transaction, the slave copies the bus data work into the appropriate offset (*BusAddr – LOW*). Otherwise, it sets the bus data pointer to the offset into its addressable memory. It then notifies the *SlaveDone* event to signal the completion of the transaction.

8.2 Transducer FIFO Channels

The data in transit via the transducer is stored locally in the FIFO. The FIFO is divided internally into partitions. The number of such partitions is equal to the total number of routes through the transducer. The FIFO is modeled at the transaction-level by a set of built-in SystemC FIFOs (*sc_fifo*). Each partition is modeled by a separate *sc_fifo* of specified size. The data type of the FIFO is selected to be *char* to represent raw bytes of data.

Listing 5. Bus channel MemServe method

```
1: UBC::MemServe (void *Addressable) {
2: while (1) {
3:   wait AddrSet;
4:   if (BusAddr >=LOW && BusAddress <= HIGH) {
5:        if (RNW == false)
6:            memcpy (Addressable + BusAddr – LOW,
                      BusData, Size);
7:        else
8:            BusData = Addressable + BusAddr – LOW;
9:        notify SlaveDone;
10:       break;
11:  } // end if BusAddr…
12: } // end while (1)
13:} // end bus.MemServe
```

The SystemC FIFO objects provide methods for both blocking and non-blocking functions. However, as per the transducer operation semantics described in Section 5, the request manager checks the status of the FIFO before the I/O controller is allowed to read or write in the FIFO. Therefore, the only FIFO access methods that we are interested in for transducer modeling are:

1. *num_free*: status check method called by the request manager for a write transaction and returns the space available in the FIFO.
2. *num_available*: status check method called by the request manager for a read transaction and returns the size of data available in the FIFO.
3. *nb_write*: non-blocking write method called by the I/O controller to write into the FIFO after availability of space has been confirmed.
4. *nb_read*: non-blocking read method called by the I/O controller to read from the FIFO after availability of data has been confirmed.

8.3 Request Manager Module

In general, before any data is sent to, or received from, the transducer, a request must be made by the sender or receiver PE by writing the size of the data into a specified location in the request buffer. The request manager reads the buffer and schedules the requests. We model the request manager as a SystemC module as shown in Listing 6. The module consists of memory allocated in the size of the request buffer (*ReqBuf*), a thread to expose the request buffer to the PEs on the appropriate bus channel (*RBCtrl*), and a method for scheduling the requests (*GetNextReq*). The module keeps a counter for the number of active requests (*NumReq*) and an event that is notified by *RBCtrl* every time a new request is made (*NewReq*). We also define an array of pointers to FIFO channels, indexed by the route ids. The pointers are assigned to the appropriate channels in the RBCtrl during initialization (not shown).

Listing 6. Pseudo-code for Request Manager module

```
1: BusWord ReqBuf [<#routes>];
2: event NewReq;
3: int NumReq;
4: void RBCtrl() {
5:   NumReq = 0;
6:   while (1) {
7:     <BUS>.MemServe(ReqBuf, LOW_<RB>,
                          HIGH_<RB>);
8:     NumReq++;
9:     notify (NewReq);
10:  } // end while
11:} // end RBCtrl
12: int GetNextReq (int *PktSize) {
13:  int route_id;
14:  // iterate and find feasible request (route ID)
15:  while ((route_id = SelectReq() == -1)
16:     wait (NewReq | FifoTrEv);
17: *PktSize = GetPktSize(ReqBuf, route_id);
18: return route_id;
19:} // end GetNextReq
20: void Clear (int route_id) {
21:   ReqBuf[route_id] = 0;
22:   NumReq --;
23:} // end Clear
```

RBCtrl initializes the number of active requests and calls the *MemServe* of the appropriate bus to expose request buffer memory. The bus addressing of the request buffer (*LOW*<*RB*> to *HIGH*<*RB*>) is defined at transducer configuration time. Each time a new request is written into the request buffer memory, the number of requests is incremented and the *NewReq* event is notified.

The request manager module implements an interface that defines functions for scheduling the requests *(GetNextReq*) and clearing them once they have been processed (*Clear)*. The *GetNextReq* method calls the *SelectReq* method that iterates over the active requests and uses the *num_available* or *num_free* functions of the appropriate FIFO to ensure that the FIFO has enough data or space available to complete the transaction. If no such request is found, *SelectReq* returns *-1* and the *GentNextReq* method waits for the *NewReq* event or a transaction to a FIFO, indicated by notification of event *FifoTrEv*. Either of those events might result in a feasible request. Once a feasible request is found, the packet size for the transaction request is looked up and returned together with the ID of the route associated with the request (*route_id)*. A request is cleared by resetting the appropriate location in the request buffer and decrementing the number of requests.

8.4 IO Controller Module

The IO controller module implements a thread to manage data transactions through the transducer as shown in Listing 7. The thread defines a local buffer (*Pkt*) of the size of maximum data packet. Similar to request manager, an array of pointers to FIFO channels, indexed by the route ids, is defined and the pointers are assigned to the appropriate channels during initialization (not shown). The IO controller thread calls the *GetNextReq* function of the request manager to obtain the route id corresponding to the highest priority active and feasible request. It then uses a lookup table, indexed by route id, to determine the type of transaction. For a read transaction, a packet of given size is read from the appropriate FIFO. The IO controller then notifies the requesting PE that its request will be processed next. This is done by notifying the appropriate interrupt event. The interrupts are defined globally for each route id of a given transducer-bus interface. After the interrupt is sent, the IO controller exposes its local packet memory on the bus by calling the MemServe method of the appropriate UBC. The packet memory's address range [*LOW*<*IO*>, *HIGH*<*IO*>] is defined at transducer configuration time.

Listing 7. Pseudo-code for IO controller module

```
1: void IOCtrl() {
2:   char Pkt [MAX_PKT_SIZE];
3:   int route_id, pkt_size, i;
4:   while (1) {
5:     route_id = RM.GetNextReq(&pkt_size);
6:     if (TrType[route_id] == READ);
7:         for (i=0; i<pkt_size; i++)
8:           fifo[route_id]→nb_read(Pkt+i);
9:     notify (Interrupt_<BUS>_<TX>[route_id]);
10:   <BUS>.MemServe(Pkt, LOW<IO>, HIGH<IO>);
11:    if (TrType[ route_id] == WRITE);
12:         for (i=0; i<pkt_size; i++)
13:           fifo[route_id]→nb_write(Pkt+i);
14:   notify (FifoTrEv);
15:   RM.Clear(route_id);
16:  } // end while
17:} // end IOCtrl
```

Conversely, for a write transaction throught the transducer, the IO controller first interrupts the requesting PE, thereby indicating that the PE may write to packet memory. It then exposes its local packet memory on the bus. Once the packet memory is written, it copies over the packet into the appropriate FIFO. Finally, the *FifoTrEv* event is notified to potentially wake up the *GetNextReq* method in a different request manager module, and the current request is cleared.

The software interface to the transducer TLM is designed similar to the final software implementation, shown in Listing 1. The interrupt event wakes up the requesting PE, which is waiting on the interrupt after writing the request (see Listing 1, lines 5-6). The PE then reads or writes the packet from/to the IO controller's memory over the UBC to complete the transaction at its end.

9 Automatic Transducer TLM Generation

Once the TLM structure and semantics are well defined, it can be generated automatically from the system level specification shown in Figure 1. We have developed automatic TLM generation methods and their C++ implementation in a tool [20]. In this section, we briefly describe methods for automatic generation of the different SystemC contructs used to model the transducer.

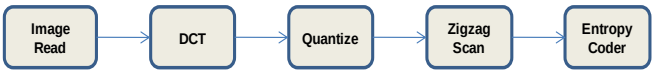

Fig. 10. Block diagram for JPEG encoder.

For each transducer in the platform, we create a new SystemC module with ports of type UBC slave interface. The number of such ports depends on the number of transducer-bus connections in the platform definition. Inside the transducer module, we instantiate a unique SystemC FIFO channel for each route through it. Next, we define the FIFO bus access mutex and the *FifoTrEv* event.

An internal lookup table is created in the tool to maintain the correlation between the buses, transducers, route ids, global routes and the FIFO channel instances. Therefore, the lookup table can be used to locate the FIFO channel instance for a given route id corresponding to a transducer-bus interface. The lookup table is used to generate the macros corresponding to route ids. Next, we create and instantiate a request manager module and an IO controller module for each transducer-bus interface. The array of FIFO point and assignment of the pointers to channels, as described in Section 8, is also generated using the lookup table. At the global scope, we generate a set of event array definitions for the interrupts from the transducer. Each array corresponds to a unique bus-transducer interface. The size of the event arrays is determined by the number of route ids, also determined from the lookup table. Finally, we instantiate the transducers and bind their bus ports to appropriate UBC instances at the top level.

10 Experimental Setup

In this section, we present the experimental setup to measure the design quality of the transducers and the productivity gains using automatic transducer model generation. We use two industrial size examples: the JPEG encoder and the MP3 decoder, as benchmarks for our evaluation. The applications were mapped on various MPSoC platforms to generate transducers with different configurations. The designs were implemented on the Xilinx Virtex-II device and all measurements are done on the ML402 board. For functional validation, we generated TLMs for the selected mappings. The TLMs were compiled and executed on a 2 GHz x86 host running Linux.

10.1 Applications

Figure 10 shows the block diagram of JPEG encoder [21]. The JPEG encoder is a pipelined model consisting of five tasks. The encoder first partitions the input bitmap image into 8x8 blocks of pixels and the blocks are applied to a 2-dimensional Discrete Cosine Transform (DCT). Next, the transform matrix is normalized by an 8x8 quantization matrix and the quantized DCT coefficients form a matrix. The elements of the matrix are ordered in a zigzag scan. Then, an entropy coder combined with a run-length coding of the zeros generates an efficient representation of the quantized coefficients to be transmitted or stored.

Figure 11 shows the block diagram for the MP3 decoder application [22]. The Huffman decoder block, *HuffDec*,

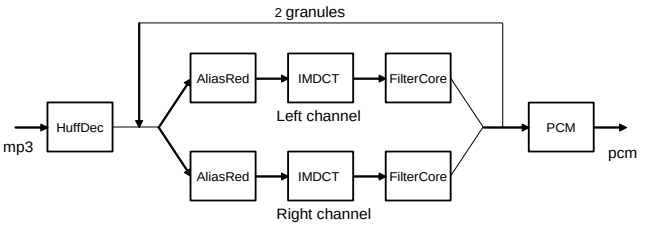

Fig. 11. Block diagram for MP3 decoder.

reads the codewords in the MP3 frames, translates them to symbols using variable length decoding algorithm, requantizes the symbols and reorders them. Then, the decoded frequency line is sent to alias reduction and finally, the pulse code modulated output samples are produced by applying the Inverse Modified DCT (*IMDCT*) and DCT (*FilterCore*) transforms. Since the decoder is taking in a stereo input, we have IMDCT and DCT tasks for both left and right channels.

10.2 MPSoC Platforms and Mapping

We implemented the methods described in Sections 7 in an automatic transducer model generation tool, written in C++. The tool was used along with the Embedded Design Kit (EDK) [17] from Xilinx to create five designs for the JPEG encoder and three designs for the MP3 decoder as part of our design space exploration. Figure 12 shows figures of the designs with the task mapping on PEs. Transducers in the designs are labeled as Tx. For functional validation, we implemented the TLM generation methods described in Section 9 in a separate tool. The generated TLM was compiled with SystemC libraries and executed on the host for all the designs.

For JPEG, the base design is 1a (not shown) that contains only one Microblaze processor [23]. The JPEG application can be easily pipelined and fits in the on-chip memory. Therefore, we can use a homogeneous MPSoC design to optimize its implementation. Design 1b maps the JPEG tasks to 5 difference Microblaze CPUs, all connected to a single Open Peripheral Bus (OPB). In design 1c, we use multiple buses to minimize the arbitration delays for concurrent transactions between the CPUs. The DCT is the most compute intensive task, so we further optimized the pipelined design by splitting the DCT into two tasks: DCT1 and DCT2, thereby balancing the load of the tasks. Platforms 1d and 1e are equivalent to 1b and 1c, respectively, except with 6 cores.

For MP3, the base platform is 2a (not shown) that uses only one Microblaze core with 32K cache and off-chip SRAM as main memory, since the program and data are too large to fit on-chip. The IMDCT and DCT tasks are the most time consuming, but, unlike JPEG, MP3 does not lend itself well to pipelining due to difficulties in balancing the task loads. A better alternative is to use a heterogeneous MPSoC platform with dedicated hardware accelerator cores for the IMDCT and DCT tasks as shown in Figure 12 (designs 2b and 2c).

The hardware accelerators have their own Double Handshake Bus (DHB) protocol. Platform 2b uses one instance of DCT and IMDCT, each, to accelerate the left channel decoding, while the remaining MP3 code is

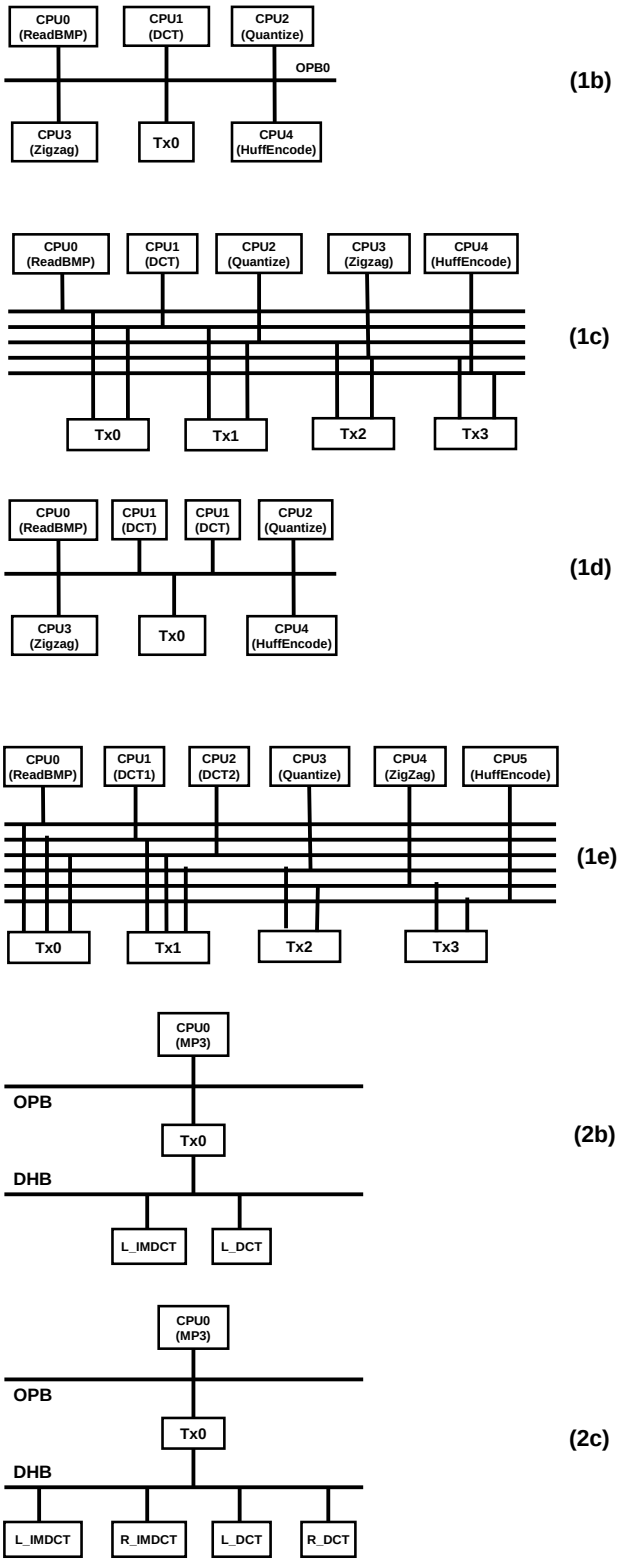

Fig. 12. MPSoC designs of JPEG encoder and MP3 decoder.

mapped to a Microblaze core. Platform 2c maps both the left and right IMDCT and DCT to accelerators.

11 Transducer Quality

We measure the design quality of the generated transducers using metrics of area and performance. The area is measured in terms of the *Slices* and Block RAMs (*BRAMs*) used in the FPGA implementation. The performance is measured in terms of the minimum cycle time obtained after synthesis and the number of cycles used for communication in the different MPSoC designs. We compare the quality of the generated transducer to manual designs of the transducer. Finally, we also present a comparison of the communication latency in the MPSoC designs that use automatically generated transducers to those with theoretical minimum communication latency.

TABLE 1
Transducer Quality Comparison to Manual Design

	Design	Slices	BRAMs	Clk(ns)
Manual	1b	722	64	25.55
	1c	3840	76	23.04
	1d	1912	72	26.12
	1e	5986	84	25.87
	2b	1616	120	27.78
	2c	2400	266	29.41
Auto	1b	682 (5.54%)	64	27.47 (-7.51%)
	1c	3526 (8.18%)	76	24.48 (-6.25%)
	1d	1689 (11.66%)	72	28.74 (-10.03%)
	1e	5468 (8.65%)	84	27.06 (-4.60%)
	2b	1444 (10.64%)	120	30.30 (-9.07%)
	2c	1790 (25.42%)	266	31.25 (-6.26%)

11.1 Comparison to Manual Design

Table 1 illustrates the design quality of automatically generated transducers using cost and performance. The transducer designs are compared to manual transducer implementations done by a PhD student with over 4 years of industrial experience in Verilog RTL development. The manual designer was given the exact same specification as given to the automatic TLM generation tool. The top level transducer block diagram, consisting of request manager, bus interface, FIFOs and IO controllers was also provided. For each of the designs (1b, 1c, 1d, 1e, 2b, and 2c), we compare the area and minimum cycles time for the transducer implementation. Designs 1a and 2a are software implementations without transducers.

The area of the transducers is measured by the number of FPGA Slices and block-RAMs (BRAMs) used by the synthesized transducer logic. The third column shows the number of slices used for transducer. The number of BRAMs used is shown in the fourth column. The cycle time is shown in the fifth column.

The second half of the tables shows the quality metrics of the automatic implementation. The difference, in percentage improvement or percentage degradation, compared to the manual design metrics is noted in parantheses. For instance, the automatically generated transducer for design 1b had 5.54% fewer slices compared to manual implementation of 1b, which is an improvement in quality. On the other hand, the minimum clock cycles time of

TABLE 2
PERFORMANCE COMPARISON TO POINT-TO-POINT LINKS

Design	Min. Cycles	Actual Cycles	Overhead (%)
1b	0.74M	0.97M	31.08
1c	0.74M	0.81M	**9.46**
1d	0.92M	1.24M	34.78
1e	0.92M	0.98M	**6.52**
2b	0.07M	0.09M	28.57
2c	0.15M	0.16M	**6.67**

automatically generated transducer in design 1b was 7.51% longer compared to manual implementation, which is degradation in quality.

While the overall structure of the transducer implementations was similar for the two implementations, we discovered marked differences in the lower level implementations. In particular, the manual designer was prone to minimizing clock cycle time by adding additional buffer registers. For instance, the critical path in the transducer was the scheduler inside the request manager. The additional registers in the manual design reduced the clock cycle time, but manifested themselves as higher number of slices needed for the implementation. The block RAMs used for FIFO implementation remained the same for both manual and automatically generated designs because of well defined FIFO sizes and structure.

Overall, we found that the automatically generated transducer was close to manual design in quality, with a worst case difference of only 10% in clock speed. The modules were clocked by the OPB bus clock which had a minimum period of 30 ns. As such, the actual throughput differences between the manual and automatically generated transducers were even smaller. The next section provides key metrics on transducer performance.

11.2 Comparison to Point-to-Point Links

Table 2 shows a comparison of the transducer performance to the theoretical optima for each design. The second column (Min. Cycles) shows the theoretical minimum communication cycles (in millions) in the hypothetical platform, when all PEs are connected to each other with unbounded FIFO buses. Therefore, the only communication time spent is in doing the bus read/writes. Such a fully-connected point-to-point communication architecture may be feasible, provided all PEs have enough ports and have compatible interface protocols. Even so, such a design would be impractical due to the high density and huge power consumption of interconnects. However, such a design is a good benchmark to compare the performance of transducer-based designs.

The third column shows the actual total number of cycles spent (in millions) for communication by all the PEs in the transducer based designs. As we can see in the fourth column, the overhead of the transducer decreases significantly as the number of PEs and buses in the system increase. The exceptions are platforms 1b and 1d, which use a single shared bus for all the PEs. In these platforms, the arbitration delay causes the majority of the overhead because several concurrent transactions collide on the shared bus. The overheads resulting from request management and FIFO size bounds can be seen, without the impact of bus arbitration, in platforms 1c and 1e. As we can see, this overhead is below 10%, thereby demonstrating the efficiency of the transducer.

In the case of design 2b, the overhead is relatively large due to several transactions of small packets through the transducer. Therefore, the request management and synchronization contributed a large proportion of the delay. In contrast, design 2c has a smaller overhead because the transaction between CPU0(MP3) and L_IMDCT/L_DCT overlaps in time with the transaction between CPU0(MP3) and R_IMDCT/R_DCT, thereby reducing the impact of synchronization and request management.

TABLE 3
PRODUCTIVITY GAIN IN RTL DEVELOPMENT

Design	Verilog LoC	Manual Time (hrs)	Generation Time (sec)
1b	3368	67	0.715
1c	13990	280	2.178
1d	3720	74	0.718
1e	15904	318	2.743
2b	1974	39.5	0.584
2c	2806	56	0.612

12 PRODUCTIVITY GAIN

Automatic generation of transducer RTL models helps with improving the productivity of performance validation and design implementation by eliminating the time consuming and error prone task of manual design. By the same token, automatic TLM generation greatly improves the productivity of functional validation.

Table 3 shows the productivity gain using automatic transducer RTL model generation. The first column lists the various designs we have created. Note again that 1a and 2a are missing since they do not involve any transducers. The second column shows the Verilog lines of code (LoC) automatically generated for the different designs. The LoC increases with the number of transducers, as seen in 1c and 1e. The third column lists the number of hours it took to code, debug and test the transducer RTL designs (m). Note that we have used cumulative person-hours here. For instance, the transducer for 1d was build on top of existing design for 1b, because the two designs have similar structure. It took 7 additional hours (74-67) to derive the transducer in 1d from the one in 1c. Transducers in 1e were derived from those in 1c and transducer in 2c was derived from the one in 2b.

The fourth column shows transducer generation time (t) in seconds. Transducer models for complex MPSoC architectures with 6 PEs, and as many buses, are generated in the

TABLE 4
DESIGN SPACE EXPLORATION WITH AUTOMATIC MODELING

Design	Spec. Time(hrs)	FPGA Impl. Time (hrs)	Total Cycles (millions)
1a	1	1	6.27
1b	2	5	3.68
1c	4	10	3.23
1d	2	5	2.45
1e	4	10	1.72
2a	1	1	4.99
2b	3	3	4.31
2c	5	5	4.23

order of few seconds. The transducer models can be fairly large, in the order of thousands of lines of code, which makes their manual development both error prone and time consuming.

The automatic transducer generation tool was used to explore the best MPSoC implementation for the JPEG and MP3 applications. The results of this effort are shown in Table 4. Besides the RTL generation time, we have two other factors. The first is the design specification time (d), shown in the second column, which is the time taken to partition the source code into separate tasks and to create the system level specification for transducer generation. The second factor is the bitstream generation time (b), shown in the third column, which includes the time to instantiate the platform components in Xilinx EDK and to synthesize the design, resulting in a bitstream for programming the target FPGA. We define the RTL productivity gain as the ratio of overall design time using manual transducer RTL development to one using our automatic transducer RTL generation tool. In other words,

$$RTL\ Productivity\ gain = (m+d+b)/(t+d+b)$$

Based on numbers from Tables 3 and 4, we achieved productivity gain ranging from 8.9X to 23.72X, with an average of 14.14X. Clearly, these numbers are to be expected, given the time consuming nature of manual RTL design. It must be noted that we do not take the time for design of the IP cores themselves.

The fourth column in Table 4 shows the total number of cycles needed to encode one JPEG frame or decode one MP3 frame. As expected, adding PEs to the MPSoC improves the performance to an extent. Since JPEG is a pipelined streaming application, it is easier to optimize it in an MPSoC implementation. There is a huge improvement simply by diving the tasks on different PEs, even if they share a bus (1b). Adding dedicated busses and transducers for each PE improves the performance further (1c). However, dividing the DCT into two tasks makes the pipeline more balanced, which pays off better than simply using dedicated buses (1d). Finally, by using both strategies of dividing DCT and using dedicated busses, we get the best performance (1e), which is almost 4X faster than a single PE design.

The MP3 design has fewer opportunities for parallelization, so we used hardware acceleration to speed up the compute intensive DCT and IMDCT tasks. Part of the speedup in 2b over 2a (pure software) comes from hardware acceleration of the left channel DCT and IMDCT. Additionally, the right channel DCT and IMDCT runs concurrently in the processor. Design 1c provides further improvements by concurrently accelerating the DCT and IMDCT tasks for both channels. The exploration was completed for both designs in less than two days using automatic RTL generation. With manual design, such an exploration would have taken several weeks.

TABLE 5
PRODUCTIVITY GAIN IN FUNCTIONAL VALIDATION WITH TLMS

Design	SysC LoC	Manual (hrs)	Gen. Time (s)	Sim. Time (s)	Sim. cps
1b	626	31	0.660	0.023	160M
1c	1860	93	0.703	0.045	71.78M
1d	840	42	0.665	0.029	84.48M
1e	2374	118	0.804	0.052	33.08M
2b	626	31	0.605	0.12	35.92M
2c	2095	104	0.633	0.55	7.69M

Table 5 shows the productivity gains in functional validation by using automatic TLM generation. The second column shows the total SystemC lines of code (LoC) for the transducer modules in the design. Since each transducer in the design has a different structure, we need to generate a unique SystemC module for each transducer. Hence, as the number of transducers in the design increases, the code size increases accordingly. We did not have resources to do manual TLM development, so the time taken for manual design is an estimate based on the code size of the automatically generated TLM. We have used an optimistic figure of 20 LoC per person-hour to estimate the manual coding and debugging time, as shown in the third column. As before, these are cumulative figures and do not reflect the benefits of designer experience and reuse of existing transducer models.

The transducer TLM generation time was negligible, as seen in the fourth column. This time is for the generation of the SystemC transducer module and does include the time to generate code for the remaining MPSoC components. Simulation times for one frame of input, shown in the fifth column were also under one second. A highly relevant metric of *simulation cycles per second (sim. cps)* is given for each model in the sixth column. Sim. CPS denotes the number of execution cycles that can be simulated in one second by the simulator. This figure strongly depends on the complexity of the MPSoC platform and the abstraction of the model. The numbers are derived by dividing the total number of cycles reported in Table 4 by the corresponding TLM simulation time for each design.

Functional TLM is much faster than RTL simulation on host or even instruction-set simulation models (ISS). For instance, the RTL simulation of design 2c took over 16 hours for one frame of data on the same host as the one used for TLM simulation. The ISS model took over 3

hours for the same simulation. Therefore, we can replace ISS or RTL simulation with automatically generated SystemC TLMs for functional validation. At the other end, we can use the automatically generated and synthesizable RTL models of the transducer to rapidly prototype MPSoC designs on FPGA and validate their performance at close to real-time execution speed.

13 Conclusion

In this paper, we proposed a highly flexible, scalable and efficient communication module called Transducer, which supports inter-PE communication in a bus-based MPSoC system. We defined the internal architecture of the transducer and methods for automatically generating a synthesizable model of the transducer from a system level specification. We also presented TLM semantics for the transducer, and methods for automatic TLM generateon based on the semantics. The TLMs were targeted for early functional validation of MPSoC designs. We showed that automatic transducer generation avoids the time consuming and error prone task of manual communication modeling in MPSoC systems. We also defined a software interface for the transducer, which simplifies software update after changes in the MPSoC communication architecture. The productivity gains from automatic transducer model generation and the minimal communication overhead of the transducer make a strong case of the use of transducers in MPSoC system design. For our future work, we are investigating methods for automatically optimizing the transducer parameters based on application profile.

Acknowledgment

This work was supported by the Center for Embedded Computer Systems, UC Irvine, while the authors were affiliated with it. We would like to thank Prof. Daniel Gajski for his vision, support and guidance, and Dr. Pramod Chandraiah for providing the MP3 application.

References

[1] Networks on Chips: Technology and Tools, G. De Micheli and L. Benini, Eds. San Mateo, CA: Morgan Kaufmann, 2006

[2] OSCI[online]. Available: http://www.systemc.org

[3] TLM 2 Whitepaper. May 2007. http://www.systemc.org

[4] Donlin, "Transaction level modeling: flows and use models." CODES+ISSS, 2004, pp 75-80.

[5] L. Cai and D. Gajski, "Transaction level modeling: an overview." ISSS-CODES, pp 19-24, 2003.

[6] A. Gerstlauer, D. Shin, J. Peng, R. Dömer, D. Gajski, "Automatic, Layer-based Generation of System-On-Chip Bus Communication Models," IEEE TCAD, vol. 26, no. 9, pp. 1676-1687, September 2007.

[7] Cornet, J., Maraninchi, F., Maillet-Contoz, L., "A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip," *DATE,* pp.9-14, 2008.

[8] Chen Kang Lo, Ren Song Tsay, "Automatic generation of Cycle Accurate and Cycle Count Accurate transaction level bus models from a formal model," *ASP-DAC,* pp.558-563, 2009.

[9] Van Moll, H.W.M., Corporaal, H., Reyes, V., Boonen, M., "Fast and accurate protocol specific bus modeling using TLM 2.0," *Design, Automation & Test in Europe Conference,* pp.316-319, 2009.

[10] Bombieri, N., Fummi, F., Guarnieri, V., "Automatic synthesis of OSCI TLM-2.0 models into RTL bus-based IPs," *High Level Design Validation and Test Workshop (HLDVT),* pp.105-112, 2010.

[11] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," *ISSS-CODES*, pp. 9–14, 2007.

[12] Gladigau, J., Gerstlauer, A., Haubelt, C., Streubühr, M., Teich, J., "A system-level synthesis approach from formal application models to generic bus-based MPSoCs," *International Conference on Embedded Computer Systems (SAMOS), 2010* pp.118-125, 2010.

[13] Grasset, A., Rousseau, F., Jerraya, A.A. , "Automatic generation of component wrappers by composition of hardware library elements starting from communication service specification," *Rapid System Prototyping*, pp. 47- 53, 2005.

[14] Zimmermann, J., Bringmann, O., Braun, A., Rosenstiel, W., "Integration of high-level synthesis in ESL platform modeling by automated generation of protocol adapters," *International Conference on Communications, Circuits and Systems*, pp.1149-1154, 2009.

[15] Watanabe, S., Seto, K., Ishikawa, Y., Komatsu, S., Fujita, M., "Protocol Transducer Synthesis using Divide and Conquer approach," *ASP-DAC*, pp.280-285, 2007.

[16] Alberto Sangiovanni-Vincentelli, "Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design," *Proceedings of the IEEE*. 95(3), pp. 467-506, March 2007.

[17] Xilinx. Embedded System Tools Reference Manual. 2005.

[18] H. Cho, S. Abdi, "Automatic generation of transducer models for multicore system design," *IEEE HLDVT*, pp.72-79, 2011.

[19] S. Abdi, G. Schirner, I. Viskic, H. Cho, Y. Hwang, L. Yu, D. Gajski, "Hardware-dependent software synthesis for many-core embedded systems," ASP-DAC, pp. 304-310, 2009

[20] S. Abdi, Y. Hwang, L. Yu, G. Schirner, D. Gajski, "Automatic TLM Generation for Early Validation of Multicore Systems," in IEEE Design & Test of Computers 28(3), pp 10-19, 2011.

[21] G. K. Wallace, "The JPEG still picture compression standard," Commun. ACM, Vol. 34, pp. 30-44, 1991.

[22] ITU-T, ISO/IEC JTC1, "Information technology - Generic coding of moving pictures and associated audio information," ITU-T Recommendation ISO/IEC 13818-3, 1998.

[23] Xilinx. MicroBlaze Processor Reference Manual. 2007.

Hansu Cho received his Ph.D. in Electrical and Computer Engineering from the University of California, Irvine in 2009. He is currently a senior engineer at Samsung DMC R&D Center in Korea. His research interests include system level design and optimization and application specific instruction-set processor design. He is a member of the IEEE.

Lochi Yu received his Ph.D. in Electrical and Computer Engineering from the University of California, Irvine in 2009. He is currently an Associate Professor in the Department of Electrical Engineering at the University of Costa Rica. His research interests are in Embedded Systems and Bio-electrical signal processing. He is a member of the IEEE.

Samar Abdi received his Ph.D. in Information and Computer Science from the University of California, Irvine in 2005. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Concordia University, Montreal. His research interests are in embedded system modeling, synthesis and verification. He has published a book on Embedded System Design and over 30 conference and journal papers. He is a member of the IEEE.