System-Level Communication Modeling for Network-on-Chip Synthesis

Andreas Gerstlauer, Dongwan Shin, Rainer Dömer, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine, USA
{gerstl,dongwans,doemer,gajski}@cecs.uci.edu

Abstract— As we are entering the network-on-chip era and system communication is becoming a dominating factor, communication abstraction and synthesis are becoming the integral part of system design flows. The key to the success of any design flow are well-defined abstraction levels and models, which enable automation of early validation, synthesis and verification. In this paper, we define system communication abstraction layers and corresponding design models that support successive, stepwise refinement from abstract message-passing down to a cycle-accurate, bus-functional implementation. Experimental results show the benefits of our definitions and design flow.

I. Introduction

As SoCs grow in complexity and size, on-chip communication is becoming increasingly important. Furthermore, new classes of optimization problems arise as communication delays and latencies across the chip start dominating computation delays. In other words, simple (e.g. bus based) communication architectures are not sufficient any more. Therefore, as we enter the network-on-chip (NoC) era, new network-based communication architectures and design flows are needed.

Communication design for SoCs poses unique challenges in order to cover a wide range of architectures while at the same time offering new opportunities for optimizations based on the application-specific nature of system designs. The goal is therefore, to develop a corresponding NoC communication design flow that enables rapid design space exploration through design automation in order to achieve the required productivity gains while supporting a wide range of implementations.

In order to automate the NoC design process, a well-defined design flow with clear and unambiguous abstraction levels, models, and transformations is required. The key to the success of this approach are properly defined design models. Arbitrary models without clear semantics do not enable synthesis and verification. For example, only subsets of hardware description languages such as VHDL or Verilog are synthesizable or verifiable. In addition, synthesis requires clear definitions of the target architecture and the set of synthesis steps to transform the input model into the target model.

In this work, we aim to define such models, design steps, and corresponding model transformations that are necessary for an automated network-on-chip design flow. Note that due to space limitations, this paper can only provide an overview of the approach. Details can be found in [8].

A. Communication Design Flow

Fig. 1 shows the proposed communication design flow. Communication design starts with a virtual architecture model of the system in which processing elements (PEs) communicate via abstract channels with untimed synchronous or asynchronous message-passing semantics. In a first *network design* task, the global system network is designed and end-to-end communication between PEs is mapped into point-to-point communication between stations of the network architecture. The result of the network design step is a refined link model of the system. In the link model, PEs and other network stations communicate via logical link channels that carry streams of packets between directly connected components.

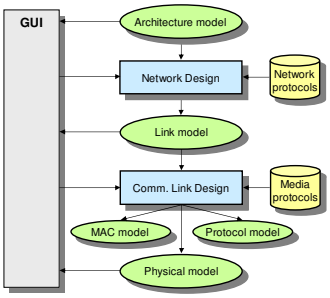

Fig. 1. Communication design flow.

In the second *communication link design* task, logical links between adjacent stations are then grouped and implemented over an actual communication medium where each group of links can be implemented separately. As a result of the communication design process, a physical model of the system is generated. The physical model is a fully structural model in which stations are connected via pins and wires and communicate in a cycle-accurate manner based on media protocol timing specifications. In the backend process, behavioral descriptions of computation and communication in each component of the physical model are then synthesized into targeted hardware or software implementations.

Apart from the physical model, the communication design flow can produce transaction-level models (TLMs) which abstract the pin-level communication in the physical model to the level of media access or individual protocol word/frame transactions. Depending on the parameters of the implementation, automatically generated TLMs can be used to trade off accuracy and model complexity for simulation speed, for example.

B. Related Work

There is a wealth of system-level design languages (SLDL) like SystemC [1] or SpecC [2] available for modeling and describing systems at different levels of abstraction. However, the languages itself do not define any details of actual concrete design flows. More recently, SLDLs have been proposed as vehicles for so-called transaction-level modeling (TLM) for communication abstraction [4]. However, no specific definition of the level of abstraction and the semantics of transactions in such models have been given. Furthermore, TLM proposals so far focus on simulation only and they lack the path to vertical integration of models for implementation and synthesis.

Layer	Interface semantics	Functionality	Impl.	OSI
Application	N/A	• Computation	Application	7
Presentation	PE-to-PE, typed, named messages • `v1.send(struct myData)`	• Data formatting	Application	6
Session	PE-to-PE, untyped, named messages • `v1.send(void*, unsigned len)`	• Synchronization • Multiplexing	OS kernel	5
Transport	PE-to-PE streams of untyped messages • `strm1.send(void*, unsigned len)`	• Packeting • Flow control • Error correction	OS kernel	4
Network	PE-to-PE streams of packets • `strm1.send(struct Packet)`	• Routing	OS kernel	3
Link	Station-to-station logical links • `link1.send(void*, unsigned len)`	• Station typing • Synchronization	Driver	2b
Stream	Station-to-station control and data streams • `ctrl1.receive()` • `data1.write(void*, unsigned len)`	• Multiplexing • Addressing	Driver	2b
Media Access	Shared medium byte streams • `bus.write(int addr, void*, unsigned len)`	• Data slicing • Arbitration	HAL	2a
Protocol	Unregulated word/frame media transmission • `bus.writeWord(bit[] addr, bit[] data)`	• Protocol timing	Hardware	2a
Physical	Pins, wires • `ADDR.drive(0)` • `DATA.sample()`	• Driving, sampling	Interconnect	1

TABLE I. COMMUNICATION LAYERS.

There are several approaches dealing with automatic generation, synthesis and refinement of communication [3, 7]. None of these approaches, however, provide intermediate models breaking the design gap into smaller steps required for rapid, early exploration of critical design issues. Furthermore, to our knowledge, there is no approach that deals with methodical and automated implementation of communication over network-oriented, non-traditional communication structures. In [6], the authors show an approach for modeling of communication at different levels of abstraction with automatic translation between levels based on message composition rules. However, they do not describe an actual design flow that includes support for arbitration and interrupt handling in traditional bus-based architectures.

II. COMMUNICATION LAYERS

The communication design flow is structured along a layering of communication functionality within each task of the design flow. The implementation of SoC communication is divided into several layers based on separation of concerns, grouping of common functionality, dependencies across layers, and early validation of critical issues for rapid and efficient design space exploration through humans or automated tools.

Table I summarizes the layers for SoC communication by listing for each layer its interface of services offered to the layer above, its functionality, and the level where it will be implemented through the backend tools (software, operating system kernel, device driver, hardware abstraction layer (HAL), hardware). Layering is based on the ISO OSI reference model [9]. However, due to the unique features and characteristics of SoC communication, layers have been tailored specifically to network-on-chip requirements. Furthermore, note that layers only serve as a specification of the desired implementation. As part of communication synthesis within each tool, layers may be merged for cross-optimizations.

A. Network Design

Network design implements presentation, session, transport, and network layers. The *presentation layer* is responsible for data formatting. It converts abstract data types in the application to untyped data blocks as defined by the canonical network byte layout. The *session layer* implements end-to-end synchronization for synchronous communication and multiplexing of channels into a set of end-to-end message streams. The *transport layer* splits messages into packets (e.g. to reduce required intermediate buffer sizes) and optionally implements end-to-end flow control and error correction. Finally, the *network layer* is responsible for routing and multiplexing of end-to-end paths over individual point-to-point links. As part of the network layer, additional communication stations are introduced as necessary, e.g. to create and bridge subnets, splitting the system of connected PEs into several segments.

B. Link Design

Link design implements link, stream, media access, and protocol layers. The *link layer* determines interface types (e.g. master/slave) and implements any necessary synchronization over underlying control and data streams. The *stream layer* multiplexes control and data streams over shared media by separating them in space (but not time) through addressing and polling. The *media access layer* is responsible for slicing data packets into protocol transactions and for regulating and separating simultaneous accesses in time (e.g. through arbitration, possibly introducing additional arbiter components). Finally, the *protocol layer* implements the timing- and pin-accurate driving and sampling of wires.

III. IMPLEMENTATION

We have implemented network and communication refinement tools that can generate design models corresponding to various communication layers automatically [10]. Given design decisions, the tools will take a virtual architecture model of the system down to its bus-functional, physical model.

A. Experiments

In order to demonstrate the modeling concepts, we applied the communication design flow to the example design of a mobile phone baseband platform. For additional examples,

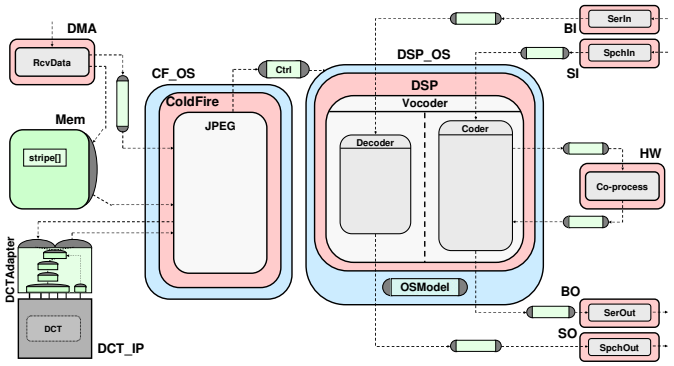

Fig. 2. Architecture model example.

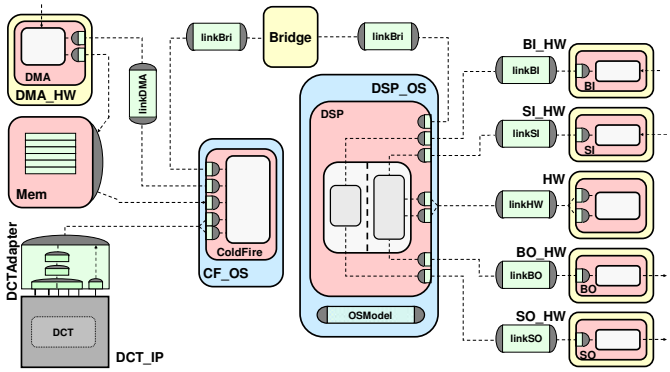

Fig. 3. Link model example.

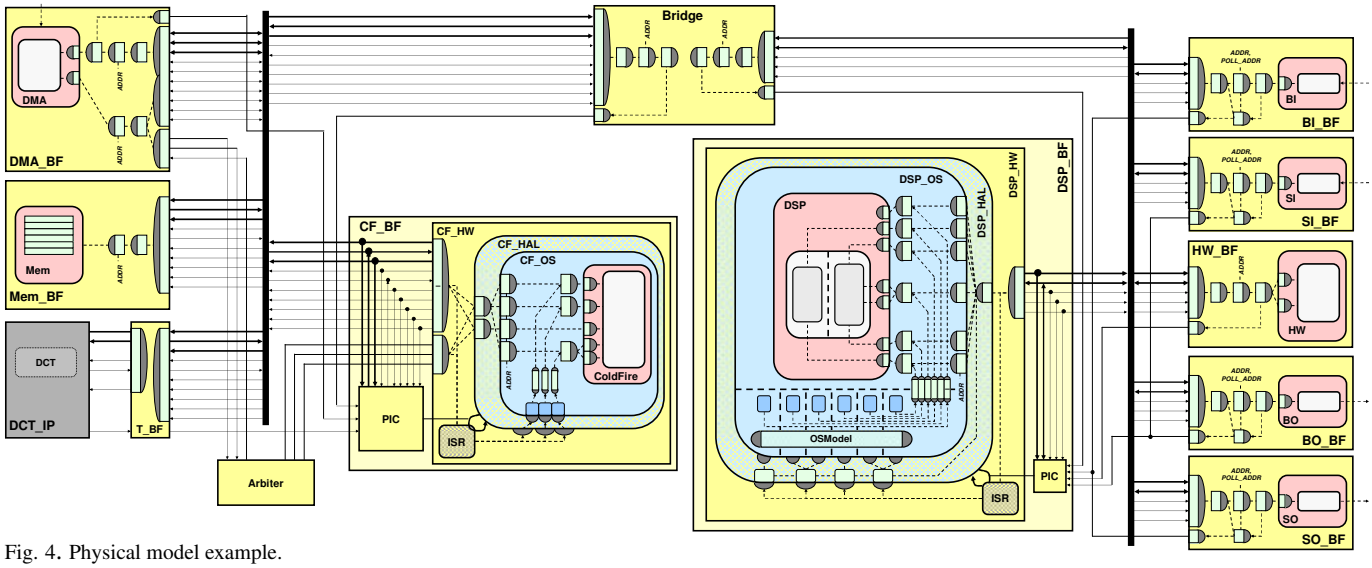

Fig. 4. Physical model example.

including application of the design flow to non-traditional, network-oriented communication architectures, see [8].

The virtual architecture model of the system at the input of communication design is shown in Fig. 2. The design consists of two subsystems: a *ColdFire* subsystem running JPEG encoding and a *DSP* subsystem for voice encoding/decoding (vocoder). The *ColdFire* processor is running the JPEG encoder in software assisted by a hardware IP component for DCT (*DCT_IP*). Under control of the processor, a *DMA* component receives pixel stripes from the camera and puts them in the shared memory (*Mem*). The *DSP* is running concurrent encoding and decoding tasks. Tasks are dynamically scheduled under the control of an operating system model [5] that sits in an additional OS layer *DSP_OS* of the DSP processor. The encoder on the DSP is assisted by a custom hardware coprocessor (*HW*) for the codebook search. Furthermore, four custom hardware I/O processors perform buffering and framing of the vocoder speech and bit streams. In the architecture model, hardware and software processors communicate via asynchronous message-passing channels.

As a result of the network design process, the network is partitioned into one segment per subsystem with a *Bridge* connecting the two segments (Fig. 3). Individual point-to-point logical links connect each pair of stations in the resulting link model. Application channels are routed statically over these links where the *Ctrl* channel spanning the two subsystems is routed over two links via the intermediate bridge. In the resulting link model, presentation layers are instantiated inside each system component. The presentation layer for communication with the DCT IP is inlined from the wrapper into the *ColdFire* processor. The memory component is replaced with a model describing the memory byte layout and presentation layers accessing the memory perform the necessary conversions of variables into memory bytes. Session, transport, and network layers are not implemented and presentation layers are routed over links through proper connectivity.

During link design, links in each subsystem are implemented over its shared medium. The native ColdFire and DSP processor busses are selected as communication media. Within each segment, unique bus addresses and interrupts for synchronization are assigned to each link and memory. In the resulting physical model (Fig. 4), link, stream, media access and protocol layers are instantiated inside the OS and hardware layers of each station. Inside the processors, interrupt handlers that communicate with link layer adapters through semaphores are created. Interrupt service routines (*ISR*) together with models of programmable interrupt controllers (*PIC*) model the processor's interrupt behavior and invoke the corresponding handlers when triggered. Components are connected via pins and wires driven by the protocol layer adapters. On the ColdFire side, an additional arbiter component regulates bus accesses between the two masters, *DMA_BF* and *CF_BF*. Finally, a transducer *T_BF* is inserted to translate between the *DCT_IP* and ColdFire bus protocols.

Model	ColdFire subsystem			DSP subsystem			System	
	Lines of code	Simulation time	Comm. delays	Lines of code	Simulation time	Comm. delays	Lines of code	Simulation time
Application	3,729	0.29 s	0 ms	12,528	17.8 s	0 ms	14,363	34.1 s
Link	3,978	0.30 s	0 ms	12,480	18.7 s	0 ms	14,535	35.2 s
Stream	4,099	0.62 s	0.28 ms	12,558	18.8 s	0.29 ms	14,754	58.4 s
Media Access	4,337	0.99 s	0.40 ms	12,782	25.2 s	0.57 ms	15,244	90.5 s
Protocol	5,313	8.66 s	1.18 ms	12,966	56.1 s	0.79 ms	16,436	544 s
Physical	5,906	20.6 s	1.50 ms	13,245	178 s	0.92 ms	17,335	1,824 s

TABLE II. EXPERIMENTAL RESULTS.

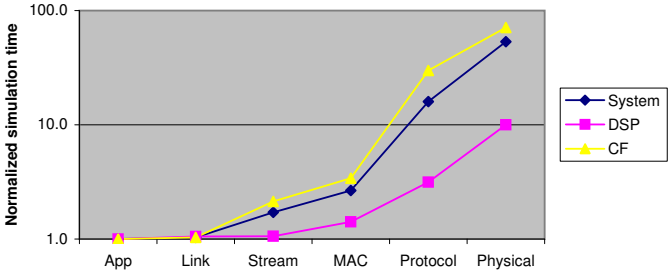

Fig. 5. Simulation performance.

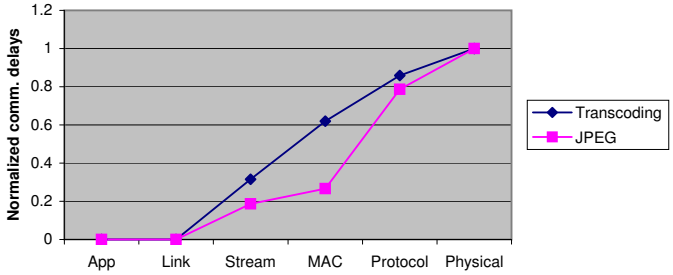

Fig. 6. Simulated communication overhead.

B. Results

Table II summarizes the results for the example design. Using the refinement tools, models of the example design were automatically generated within seconds. A testbench common to all models was created which exercises the design by simultaneously encoding and decoding 163 frames of speech on the vocoder side while performing JPEG encoding of 30 pictures with 116x96 pixels. Models of the whole system and each subsystem were simulated on a 360 MHz Sun Ultra 5 workstation using the QuickThreads version of the SpecC simulator.

Fig. 5 plots simulation times normalized against the architecture model times. Contributions of communication overhead to the simulated overall transcoding (back-to-back encoding and decoding) and encoding delays in the vocoder and JPEG encoder, respectively, are shown in Fig. 6. Delays are normalized against the overhead in the final physical model.

Results show that with increasing implementation detail at lower levels of abstraction, accuracy improves linearily while model complexities grow exponentially. Results confirm the choice of the link model as the intermediate model in the design flow that allows fast validation of the overall network topology. By definition, all models above the physical model are TLMs in which communication is abstracted away from pins and wires. The results show that depending on the architecture, MAC or protocol TLMs return accurate results at much higher simulation speeds. If there is no bus contention, the MAC model provides fast and accurate feedback. However, in the presence of arbitration, slicing of data into bus words/frames needs to be modeled in order to get accurate results that include effects of interleaved media accesses at the protocol level. In these cases, only the protocol model can provide correct delays with significantly reduced simulation speeds. Finally, at the communication level, pin- and timing-accurate results are available at the expense of huge runtimes.

IV. SUMMARY & CONCLUSIONS

In this paper, we presented a communication design flow with well-defined design steps and design models. Starting from a virtual architecture model with abstract message-passing communication, a design is brought down to a bus-functional implementation through network and link design tasks. Using an industrial-strength example, the feasibility and benefits of the approach have been demonstrated.

Out of all possible models, intermediate models have been defined based on accuracy vs. simulation speed tradeoffs allowing early validation of critical design decisions. In between design tasks, the link model defines the implementation of the end-to-end network on top of point-to-point logical links. Furthermore, two transaction-level models have been identified for providing accurate results above the pin level.

In general, models at various levels of abstraction have been defined such that they can be automatically generated through successive refinement. Therefore, the flow supports high-level communication abstractions for fast feedback and early simulation together with an automated path to implementation. In conclusion, the models are the enabler for rapid, early design space exploration and significant productivity gains.

Future work includes adding algorithms for decision making to provide a completely automated synthesis process. Furthermore, we plan to extend design tasks and refinement tools to implement error-correction, flow control, and dynamic routing for long-latency, error-prone network communication media.

REFERENCES

[1] T. Grötker et al. *System Design with SystemC*. Kluwer, 2002.

[2] A. Gerstlauer et al. *System Design: A Practical Guide with SpecC*. Kluwer, 2001.

[3] W. O. Cesário et al. "Multiprocessor SoC platforms: A component-based design approach." *IEEE D&T*, 19(6), November/December 2002.

[4] M. Coppola et al. "IPSIM: SystemC 3.0 enhancements for communication refinement." In *DATE*, 2003.

[5] A. Gerstlauer et al. "RTOS Modeling for System Level Design." In *DATE* 2003.

[6] R. Siegmund and D. Müller. "SystemC^{SV}: An extension of SystemC for mixed multi-level communication modeling and interface-based system design." In *DATE*, 2001.

[7] K. van Rompaey et al. "CoWare: A design environment for heterogeneous hardware/software systems." In *Euro-DAC*, 1996.

[8] A. Gerstlauer. "Communication Abstractions for System-Level Design and Synthesis." Technical Report CECS-TR-03-30, UC Irvine, 2003.

[9] International Organization for Standardization. *Reference Model of Open System Interconnection*, 1994. ISO/IEC 7498 Standard.

[10] S. Abdi et al. "Automatic Communication Refinement for System Level Design." In *DAC* 2003.