Automatic Generation of Bus Functional Models from Transaction level Models

Dongwan Shin, Samar Abdi, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine, CA 92697 USA
e-mail: {dongwans, sabdi, gajski}@cecs.uci.edu

Abstract– This paper presents methodology and algorithms for generating bus functional models from transaction level models in system level design. Transaction level models are often used by designers for prototyping the bus functional architecture of the system. Being at a higher level of abstraction gives transaction level models the unique advantage of high simulation speed. This means that the designer can explore several bus functional architectures before choosing the optimal one. However, the process of converting a transaction level model to a bus functional model is not trivial. A manual conversion would not only be time consuming but also error prone. A bus functional model should also accurately represent the corresponding transaction level model. We present algorithms for automating this refinement process. Experimantal results presented using a tool based on these algorithms show their usefulness and feasibility.

I. Introduction

The transaction level modeling (TLM) [7] is high-level approach to modeling the systems where details of communcations among system components are separated from the detail of the implementation of system components and communication architecture. TLM aims at communication modeling so as to optimize simulation speed. TLM has been considered to address needs for early architecture exploration and embedded software development. However, much work still needs to be done to formalize the system level design methodology and to adopt TLM in system level design flow.

In system level design, communication synthesis requires extensive design space exploration for communication architecture. With a greater number and variety of components being put together on a chip, the task of communication synthesis becomes more complicated. In order to choose the right communication architecture for our designs, we need to generate models that reflect the communication architecture. These models are then evaluated through simulation to test "goodness"of the design decisions.

Typically, these models are handwritten, which poses a number of problems. First of all, a lot of time is spent in writing these models which is a serious handicap to the exploration process. The fewer architectures we test, the lower is the probability of choosing the optimal one. Secondly, model rewriting is an error prone process. It is possible to introduce several errors while manually rewriting the model. This makes the evaluation of our communication architecture questionable.

In this paper we look at how we speed up the communication synthesis process by enabling automatic model refinement. The rest of the paper is organized as follows. Section II is a brief review of the related work in this area. Section III talks about our communication refinement flow. Section IV looks at tasks of communication refinement for the system with dynamically scheduled tasks. Finally, we present experimental results in section V and concludes this paper.

II. Related Work

In recent years, a lot of attention has been given to modeling and synthesis of bus architectures [10] [9]. CoWare [5] can support shared memory among heterogeneous processors but focuses on rendezvous communication protocol based on message passing. Jerraya et al. [8] [4] presented interesting schemes for putting together heterogeneous components on a bus using wrappers for design of application-specific multiprocessor SoCs. Grötker et al. talked about transaction level modeling in [7] that aims at communication modeling so as to optimize simulation speed. However, they do not address automatic refinement of a transaction level model to produce a timing-accurate and pin-accurate bus functional model.

III. Communication refinement flow

Fig. 1 shows how communication synthesis is performed in our system level design methodology. We begin with a transaction level model of a system. It reflects the intended architecture of the system with respect to the components that are present in the design. Each component executes a specific behavior in parallel with other components. Communication inside a component takes place through local memory of that component, and is thus not a concern for communication refinement. Inter-component communication is point-to-point and takes place through abstract channels that support *send* and *recv* methods [6]. The second input is a protocol library that a set of channels that model the protocols of system busses. These channels provide for the standard *read/write* methods for the bus protocol.

The final input is a set of synthesis decisions by user. The decisions must input to the refinement engine using a specific format. Some typical features of the communication architecture include the choice of system busses, the mapping of abstract communication to these busses, the connectivity between components and busses etc.

With these inputs, the communication refinement tool produces a bus functional model that reflects the bus architecture of the system. In the bus functional model, the top level of the

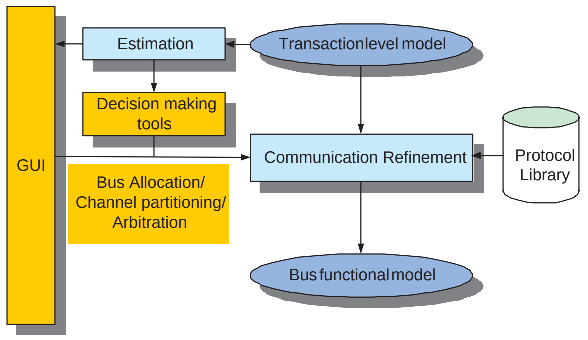

Fig. 1. Communication refinement engine

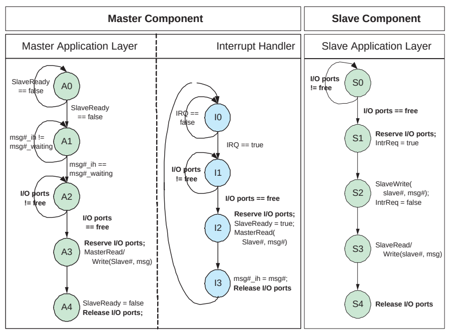

Fig. 3. Master and slave communication mechanism for simple architecture

design consists of system components and wires of the system bus(es). The components themselves are refined to their bus functional models that communicate using the system bus(es).

IV. Refinement to Communication architecture

In this section, we look at communication refinement of a simple model. We will look at the basic tasks involved in the refinement process. The design consists of two components (a processor and a HW unit) communicating with two-way blocking channels. All this communication needs to be mapped to a single system bus in order to get a simple bus architecture as shown in Fig. 2. Four communication points are shown in the master and slave component. Each communication point is labeled such that node **A** of master talks to node **A** of slave, node **B** of master talks to node **B** of slave and on. Implementation of data transactions on the system bus is done by the *application Layer* for that variable. Each component in the design has a unique *application Layer* for every variable that it sends to or receives from. The *application Layer* essentially substitutes the original abstract communication channel by implementing the data transfer on the system bus.

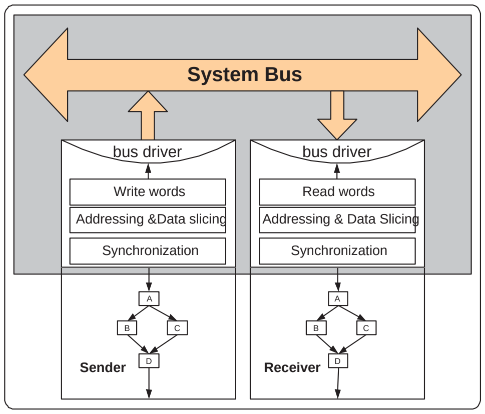

Fig. 2. Application layer for simple architecture

A. Synchronization

Besides converting abstract data to bus words, we also need to preserve the communication semantics of the transaction level model. In the case of abstract channels, each data transaction is independent and does not interfere with other transactions. However, once all those independent data transactions are mapped on the same bus, they have to share the same communication medium and synchronization events. Therefore, it is necessary to generate additional synchronization code so as to avoid conflicts on the buse [3].

If there are a set of concurrent tasks inside components, the tasks can be scheduled statically or dynamically. If the two communicating components have statically scheduled tasks, there would be no possibility of temporal overlap of communication which is well explained in [3]. In this paper, we will focus on communication between components with dynamically scheduled tasks.

With dynamically scheduled components, we are faced with a scenario where we might have temporal overlap of communication. For instance, in Fig. 2, transactions **B** and **C** might overlap in time. In such a case, we have two issues to look into.

First, we have to determine the source of the data transfer request. If the master gets an interrupt from the slave, there is no way to tell if the slave is ready for transaction **B** or **C**. To distinguish between the two transaction requests, each variable (message) should be assigned a different address. Moreover, the behavior of the interrupt handler on the master side would be shown in Fig. 3. On the master side, we also need to separate *message id* (msg#_ih) register for each message. The master is waiting for message id (msg#_waiting). The interrupt handler on receiving an interrupt event reads the message id (msg#_ih) from the registers inside the slave with slave address. After getting an acknowledge from master, the slave will put the variable address on the bus.

Second, with temporal overlap of communication, we need to control access to IO ports of the component. Therefore each data transfer has to be treated like a critical section. To ensure this, we can use the semaphore for the IO protection of the software components, and for haredware component, hardware protection mechanism like test-and-set. Note in Fig. 3 that each access to the IO ports is protected. The code generated in application layer for each component must ensure that the IO ports are reserved before they are used.

If the number of slaves is more than the number of interrupt ports on the processor's interface, we generate an interrupt controller. In case of multiple masters on a bus, arbitor needs

TABLE I
EXPERIMENTAL RESULTS FOR VARIOUS VOCODER ARCHITECTURES

Number of Components	Number of busses	Traffic/sample (bytes)	Schedule method	TLM (lines)	BFM (lines)	Modified lines (LOC)	Automatic refinement (seconds)	manual refinement (person-hr)
1 DSP56660 1 HW	1 DSPBUS	36512	static	10270	12554	2284	0.701	230
			priority-based	10307	13004	2997	0.714	300
1 DSP56660 2 HW	1 DSPBUS	46944	static	10968	14279	3311	0.739	330
			round-robin	11010	14611	3601	0.764	360
2 DSP56660 2 HW	1 DSPBUS	57276	static	11049	15373	4324	1.597	430
			priority-based	11103	16739	5636	1.687	560
2 DSP56660 3 HW	2 DSPBUS	121924	static	35309	44771	9462	3.047	950
			round-robin	35496	45557	10661	3.156	1070

to be generated. The application layer to handle interrupt and arbitration are explained in [2].

V. EXPERIMENTAL RESULTS

Based on the described methodology, we developed a CAD tool for communication refinement in C++ and integrated it into our system level design environment [1]. We tested it with the GSM Vocoder which is employed worldwide for cellular phone networks.

Different architectures using the Motorola DSP processor and custom hardware units were generated and various bus architectures were tested. Table I shows the data from tests conducted on 4 different architectures of the GSM Vocoder. The total traffic per speech sample refers to the amount of data exchanged between components during course of one simulation with a sample speech of 163 frames. Note that this data increases with greater partition, which increases communication time. We applied static and dynamic scheduling algorithm to schedule tasks in the PEs. Static scheduling removes the concurrency of tasks and serializes them in order. Dynamic scheduling is based on priority and round-robin method.

To compare against the manual effort of model refinement, we used the lines of code (LOC) metric. Even with a very optimistic estimate of 10 LOC per person hour, manual communication refinement takes several hundred hours for reasonably complex designs. Automatic refinement on the other hand completes in the order of a few seconds. The productivity gain is more than 1000 times as a result of automatic refinement even if the time for design space exploration which takes around 30 minutes to 1 hour is included for comparision.

VI. CONCLUSIONS

In this paper, we suggested a methodology algorithms to automatically generate bus functional models from transaction level model with abstract message passing semantics. A tool has been developed and experiments were performed to validate this concept. Simulations were done on input transaction level models and output bus functional models to ensure their semantic equivalence. Our main contribution in this paper is the automation of a time-consuming and error prone process to achieve better designer productivity. It also enables designers to evaluate several design points during exploration.

REFERENCES

[1] S. Abdi, J. Peng, H. Yu, et. al. System-on-chip Environment (SCE Version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-18, UC, Irvine, July 2003.

[2] D. Shin, S. Abdi, D. D. Gajski. Automatic Generation of Bus function models from Transaction level models. Technical Report CECS-TR-03-33, UC, Irvine, November 2003.

[3] S. Abdi, D. Shin, and D. D. Gajski. Automatic communication refinement in system-level design. In *Proceedings of DAC*, pages 300–305, June 2003.

[4] W. O. Cesario, A. Baghdadi, L. Gauthier, et. al. Component-baed design approach for multicore SoCs. In *Proceedings of DAC*, pages 789–794, June 2002.

[5] CoWare N2C. Available at `http://www.coware.com/cowareN2C.html`.

[6] D. D. Gajski, J. Zhu, R. Dömer, et. al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[7] T. Grötker, S. Liao, G. Martin, et. al. *System Design with SystemC*. Kluwer Academic Publishers, March 2002.

[8] D. Lyonnard, S. Yoo, A. Baghdadi, et. al. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of DAC*, pages 518–523, June 2001.

[9] R. B. Ortega and G. Borriello. Communication synthesis for distributed embedded systems. In *Proceedings of ICCAD*, pages 437–444, November 1998.

[10] T.-Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of ICCAD*, pages 288–294, November 1995.