Automatic Generation of Transaction-Level Models for Rapid Design Space Exploration

Dongwan Shin, Andreas Gerstlauer, Junyu Peng, Rainer Dömer and Daniel D. Gajski
Center for Embedded Computer Systems
University of California Irvine
CA 92697 USA
{dongwans, gerstl, pengj, doemer, gajski}@cecs.uci.edu

ABSTRACT

Transaction-level modeling has been touted to improve simulation performance and modeling efficiency for early design space exploration. But no tools are available to generate such transaction-level models from abstract input descriptions. Designers have to write such models manually, which is a tedious and error-prone task, and one of bottlenecks in improving designer's productivity. In this paper, we propose a method to generate transaction-level models from virtual architecture models where components communicate via abstract message-passing channels. We have applied our approach to a set of industrial-strength examples with a wide range of target architectures. Experimental results show that significant productivity gains can be achieved, demonstrating the effectiveness and benefits of our approach for rapid, early exploration of communication design space.

Keywords

system-level design, communication synthesis, transaction-level model

1. INTRODUCTION

As system-on-chip (SoC) designs grow in complexity and size, on-chip communication is becoming an increasingly important factor. In order to explore the communication design space, designers use models which are evaluated through simulation. Typically, these models are manually written, which is a tedious, error-prone and time-consuming process. Furthermore, to achieve required accuracies, models are written at low levels of abstraction with resulting slow simulation performance. Together, this severely limits the amount of design space that can be explored in a reasonable time.

Recently, a lot of research on defining communication models at different levels of abstractions has been proposed in order to improve simulation performance. This trend is now leveraged by the transaction-level model (TLM) paradigm, which provides system-level bus interfaces at higher levels of abstraction. In this paper, we propose an approach for automatic generation of TLMs from an abstract description of the partitioned system processing architecture.

The rest of the paper is organized as follows: We show a brief overview of related work in Section 2. Section 3 introduces the overall design flow and the inputs and outputs of the communication synthesis. Section 4 will present the details of the communication refinement process. Finally, experimental results are shown in Section 5 and the paper concludes with a summary in Section 6.

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

2. RELATED WORK

There is a wealth of system-level design languages (SLDL) like SystemC [4], or SpecC [5] available for modeling and describing systems at different levels of abstraction. However, the languages themselves do not define any details of actual concrete design flows. More recently, SLDLs have been proposed as vehicles for so-called transaction-level modeling for communication abstraction [1, 4, 6, 9]. 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 lack the path to vertical integration of models for implementation and synthesis.

Historically, a lot of work has focused on automating the decision making process for communication design [7, 8, 10] without, however, providing corresponding design models or a path to implementation. More recently, work has been done to target automatic generation [11, 15] of communication, but in all cases, the approaches are usually limited to specific target architecture templates or narrow input model semantics.

Recently, some commercial tools [2, 3] are beginning to capture designs at the transaction-level. In contrast to such existing schematic entry tools that simply provide an interface for plugging existing database models together graphically, the contribution of this paper is to generate concrete, detailed TLMs from abstract virtual architecture models of a system.

3. COMMUNICATION DESIGN FLOW

Figure 1 shows the proposed communication design flow. Design decisions are made by the user and entered into the system through a graphical user interface (GUI). With the

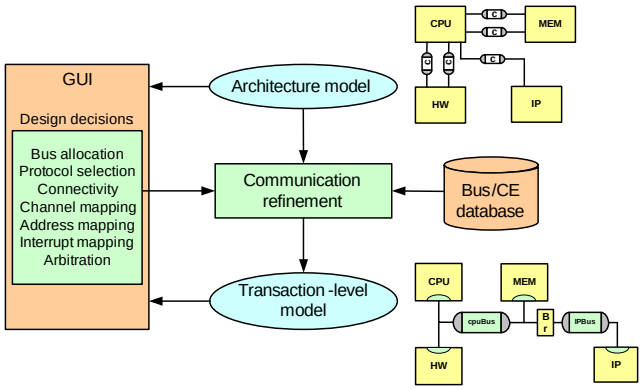


Figure 1: Communication design flow.

design decisions, the user specifies the desired target architecture and the mapping of communication onto this architecture. Based on these decisions and given the design model at the input of the flow, refinement tool synthesizes the respective implementation of the communication and generate the resulting design model at the output of the flow. In the process, refinement tool relies on a set of databases that provide models of communication elements (CEs), busses and other communication structures.

Our communication design flow starts with a virtual *architecture model* of the system in which processing elements (PEs) communicate via abstract channels. During communication synthesis, the global system network is designed and end-to-end communication between PEs is mapped into point-to-point communication between stations (PEs and CEs) of the network architecture. Then, logical links between adjacent stations are grouped and implemented over an actual communication medium. As a result of the communication design process, a *transaction-level model* of the system is generated. The TLM [1, 4, 6] abstracts pin-level communication to the level of individual bus protocol transactions in order to accelerate model simulations.

3.1 Input Architecture Model

The architecture model is the starting point for communication design. This input architecture model may be manually coded or can be automatically generated by virtual architecture generation tools [12]. It follows certain predefined semantics (see Section 4.1) and reflects the intended virtual processing architecture of the system with respect to the PEs that are present in the design. Each component in the virtual architecture is a PE that executes a specific application behavior in parallel with other PEs. Communication inside a PE takes place through its local memory. It is thus not a concern for system communication synthesis. Inter-PE communication by the application in the architecture model takes place through abstract, high-level channels of untimed message-passing or shared memory semantics.

Figure 2 shows an example of an architecture model. The application has been mapped onto a system architecture consisting of a processor (*CPU*), a custom hardware coprocessor (*HW1*), a custom hardware peripheral (*HW2*) and a system memory (*MEM*). Inside the *CPU*, tasks are dynamically scheduled under the control of an operating system model [16]. In addition to communicating via channels (*c1*, *c2* and *c3*), PEs exchange data by accessing variables (*v1* and *v2*) stored in the shared memory (*MEM*) and exported

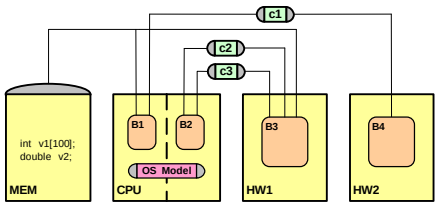


Figure 2: Architecture model example.

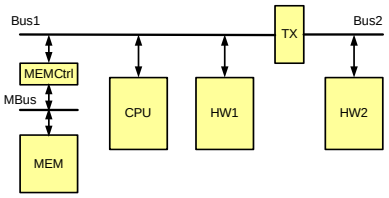


Figure 3: Target architecture for Figure 2.

by the memory through its external channel interface. At its interface, the memory provides methods to read and write the value of each variable in the memory.

3.2 User Decisions

Design decisions include allocation of system busses, protocol selection, allocation and selection of CEs (routers and bridges), definition of the connectivity between components and busses, mapping and routing of abstract communication over busses, declaration of masters and slaves, and assignment of bus addresses, interrupts, and access priorities for each message. Through the user decisions, the target architecture for the design is defined. In our case, the target architecture is limited to networks of busses in a forest of trees topology, i.e. there are no cycles.

Figure 3 shows the target architecture chosen for the previously introduced example (Figure 2). In this example, we allocate three busses: a CPU bus *Bus1* as the main system bus, a memory bus *MBus*, and a peripheral bus *Bus2*. *CPU* and *HW1* PEs are directly connected to the system bus. A memory controller CE *MEMCtrl* is allocated and connected to bridge between the system and memory bus protocols. Finally, a router CE *TX* is inserted to connect and translate between system and peripheral busses.

3.3 Databases

The databases consist of a transaction-level bus database and a CE database. The bus database contains models of busses including associated protocols. Bus models in the database implement the primitives defined by the bus protocol for data transfers and arbitration. They provide an abstraction of external communication into data links and memory accesses by using and combining bus primitives to regulate media accesses and slice abstract data into bus words. Each bus model can have two separate sides with different implementations for bus masters and bus slaves.

The CE database contains bridge and transducer components that include attributes like name, type and associated bus protocols. The models of CEs in the database, however, are empty shells that are void of any functionality. They will be synthesized by the refinement tool.

3.4 Transaction-level Model

At the output, refinement produces a TLM. The TLM accurately describes the system communication architecture

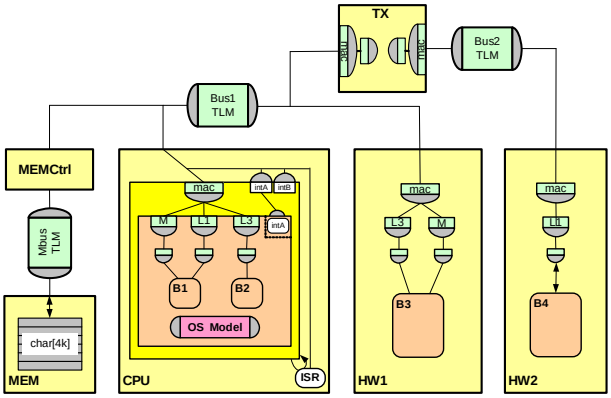


Figure 4: Transaction-level model example.

down to the level of individual bus protocol read and write transactions. In our transaction-level modeling of a system, the computation is estimated time-accurate and the communication is bus-cycle accurate (Programmer's View + Timing) [14].

Figure 4 shows the TLM generated for the input architecture model example from Figure 2. Reflecting the originally defined target architecture (Figure 3), components communicate via bus protocol transaction channels for *Bus1*, *Bus2* and *MBus*. Transaction-level implementations from the bus database together with automatically generated implementations of higher protocol stacks are inserted into each connected component. For the programmable PE (*CPU*), its transaction-level implementation is taken from the database, connected at the media access level, customized by generating appropriate interrupt handlers and filled with the PE's application code.

4. COMMUNICATION REFINEMENT

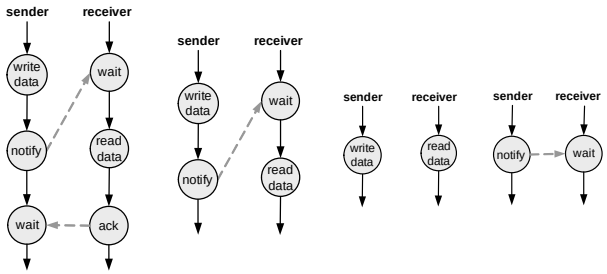
In this section, we will look at details of model transformations that are performed during refinement.

4.1 Input Communication Channels

Four different types of abstract communication are supported in an architecture model at the input of refinement: synchronous and asynchronous message passing, shared memory accesses and events. The communication semantics of these channels are shown in the form of state diagrams in Figure 5.

In synchronous message passing, as shown in Figure 5(a), both the sender and the receiver meet in a rendezvous fashion to safely exchange data. More specifically, the sender stores its data into the channel, notifies the receiver that the data is ready, and then waits for the receiver to acknowledge the receipt of the data. The receiver, on the other hand, first waits for notification of data arrival, then gets it and acknowledges the reception. In short, synchronous message passing conceptually utilizes a two-way handshake mechanism to ensure reliable data transport. This way, data cannot get lost or duplicated. However, both the receiver and the sender may be blocked in their execution.

In asynchronous message passing, as shown in Figure 5(b), only the receiver may be blocked if data is not available. The sender is not blocked. To avoid the loss of data, sent data is stored in a queue channel until it is picked up by the receiver. As a consequence, asynchronous message passing



(a) sync. MP (b) async. MP (c) memory (d) event

Figure 5: Semantics of communication channels

is also reliable, but the sender cannot make any assumptions about the point of time when the receiver actually retrieves the data.

The third type of communication, a shared memory access as shown in Figure 5(c), exhibits non-blocking communication for both the sender and the receiver. The sender simply writes data into a shared storage element from where the data can be read by the receiver at any time. Since there is no built-in synchronization between the communicating parties, this type of data transfer is unreliable. Thus, data may get lost due to overwriting, or could get duplicated due to multiple read accesses.

Finally, an event channel, as shown in Figure 5(d), exhibits pure synchronization semantics without any data transfer. Here, the receiver simply waits for an event from the sender before proceeding in its execution. No data is exchanged. In other words, event communication is non-blocking for the sender, but blocking for the receiver. Note that this event channel can be easily combined with a memory channel to achieve reliable communication in shared memory fashion.

4.2 Channel Implementation

Given the input architecture model, the communication refinement tool will automatically generate an implementation of the abstract input channels on the given communication architecture. The tool automatically generates and inserts the necessary bus drivers and bus interfaces into the system components of the system. The automatic generation of bus drivers and interfaces inside components adapts accesses from the application tasks into transactions over the bus channels connected to the components. The protocol stacks are customized and optimized in terms of (a) station type of the component on the bus (master/slave) (b) communication protocol, (c) channel semantics (Section 4.1), (d) data types of message transfered, and (e) synchronization between components.

4.2.1 *Synchronous message passing*

Synchronous message passing implements two-way block handshaking, which does not require any buffers to transfer messages between components. In order to preserve the semantics of channels in the input models, synchronization between components has to be introduced whenever necessary. In a bus-based system, we distinguish between master and slave components for each transfer. A master component will wait for synchronization from slaves before performing the actual data transfer. On the slave side, a slave will notify the master before starting to listen for incom-

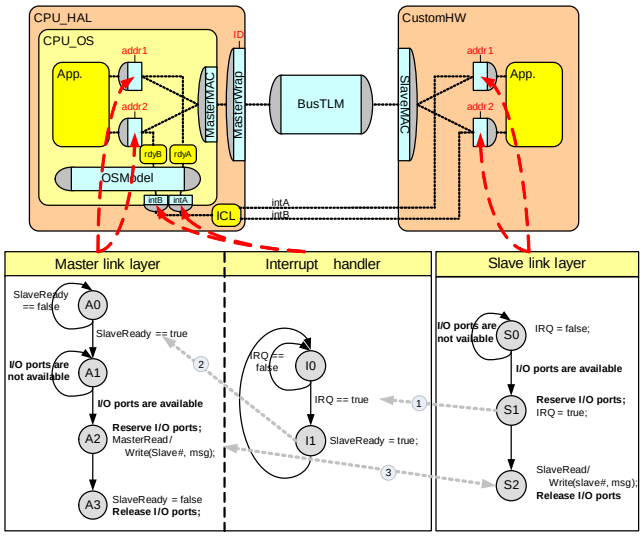


Figure 6: Synchronization by interrupts.

ing data transfer requests. In case of busses with inherent two-way synchronization built into their data transfer protocols (e.g. RS232), no explicit synchronization needs to be implemented by the refinement tool.

The synchronization from slave to master in a bus-based communication architecture has to been done through interrupts and/or polling. The decision about which synchronization mechanism to use is done by user. In case of synchronization via interrupts, the user has to assign interrupt lines for each data transfer (message passing channel).

Transaction-level models for programmable components in the PE database include a definition of their interrupt capabilities. The top-level shell defines the interrupt ports available at the physical component interface, and the hardware abstraction layer (HAL) provides corresponding empty interrupt handler templates. During model refinement, interrupt ports from slaves are connected to the interrupt ports of programmable components and interrupt handlers and interrupt tasks are generated in the HAL and operating system, respectively, by filling the corresponding templates of the processor model.

Figure 6 shows the state machines synthesized inside master and slave components which are synchronized by interrupts. When a slave process reaches the communication point, it notifies the master that it is ready to start the data transfer by sending an interrupt (1). Upon receiving the interrupt event, the master suspends its execution and the interrupt handler in the master sets a *SlaveReady* flag. The master side process waits until the flag is set (2) to initiate the bus transfer. Finally, the slave component waits for the master to initiate the bus transfer by checking the address bus (3). This mechanism retains the two-way blocking property of any original synchronous message passing communication. Once the data transfer is complete, the master component resets the *SlaveReady* flag to prepare for the next slave request.

In case of interrupt sharing due to an insufficient number of interrupts in the master, interrupt handling is extended to first determine the source of each interrupt request via polling of slaves. Due to space limitations, implementation of interrupt sharing and polling [13] is not shown here.

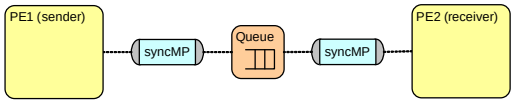


Figure 7: An implementation of async. MP.

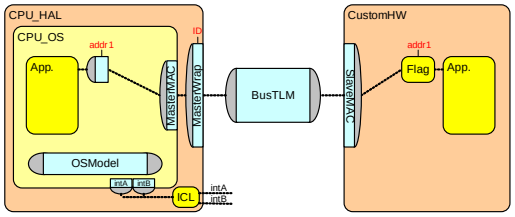


Figure 8: An implementation of event.

4.2.2 *Asynchronous message passing*

In asynchronous message passing, data is stored in a storage element such as a buffer or queue for reliable data transfers. Otherwise, the data may get lost when the receiver is not ready. Asynchronous message passing channels can have 3 types of implementations depending on which component has the storage element. Users can choose one of these implementations by assigning asynchronous message passing channels to corresponding PEs:

1. Stored in sender: sender implements the storage element from which the receiver gets data. Data transfers on the bus happen between the storage element and the receiver.
2. Stored in receiver: receiver implements the storage element. Data transfers on the bus happens between the sender and the storage element.
3. Stand-alone storage element: the storage is implemented in a separate component which buffers the data between sender and receiver (Figure 7).

As shown in Figure 7, asynchronous message passing channels are refined down to components and synchronous message passing channels in between. The synchronous message passing channels are then implemented as explained in Section 4.2.1.

4.2.3 *Memory accesses*

The abstract models of memory accesses need to be replaced with transaction-level implementations of memory components, taken out of the database. The model refinement will automatically detect corresponding implementations of memory accesses. In case of memory or register (memory-mapped I/O) accesses, slave components are assumed to be always ready and no extra synchronization is necessary. In addition, refinement tool inserts memory drivers that perform data formatting from abstract memory accesses to accesses (based on slice and offset) of the memory over the bus.

4.2.4 *Events*

Event channels are used for synchronization only. They do not carry any data. An event channel can be implemented by asynchronous message passing with a flag data (1 bit data) through which a sender notifies a receiver that the sender is ready (Figure 8). As with asynchronous message passing channels, the receiver implements the buffer to store the value of a flag on the receiver side. When an interrupt-capable processor is the receiver, event channels

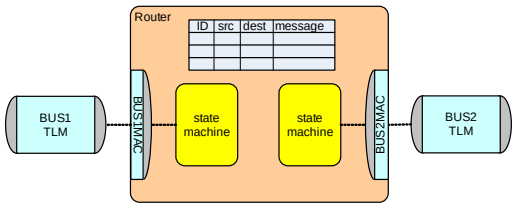
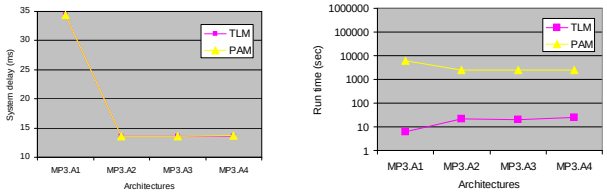


Figure 9: Routing of message in CEs.



(a) Simulated delays (b) Simulation performance

Figure 10: MP3 Exploration results.

are implemented by receiving an interrupt from the sender (similar to synchronous MP in Figure 6).

4.3 Arbitration

If a component internally has multiple tasks which access a transaction-level bus channel concurrently, the interface has to be protected to avoid potential bus conflicts of bus accesses (state A2 and S1 in Figure 6). Therefore, we implement mutual exclusiveness at the interface of the component (*MasterWrap* adapter in Figure 6 and Figure 8).

Arbitration among multiple masters on the bus is implemented as part of transaction-level bus channel. Externally, master components have to provide identity information to the bus channel. For this, the adapter inside the master interface transfers the identity (*ID* in *MasterWrap* adapter in Figure 6 and Figure 8).

4.4 Bus Bridging and Routing

If two different bus systems are connected to each other, additional CEs such as bridges and transducers are introduced. CEs split and segment the system of connected PEs in the architecture model into several bus subsystems.

Bridges transparently translate between two bus protocols directly at the protocol level. A bridge state machine is generated as the product of the two bus protocol state machines [13]. In the process, the two protocols are properly interleaved such that data dependencies and timing constraints are observed. A bridge is always a master on one side and a slave on the other. Between listening for and serving transactions on the slave side, it interleaves corresponding mirror transactions on its master side (blocking the slave side in the process, if necessary).

In cases where simple bus bridges are not sufficient, transducers are necessary. Transducers operate on packets using a store-and-forward principle, routing packets between their incoming and outgoing links. Transducers can connect any two bus protocols and they can be master or slave on either side. In contrast to a bridge, transducers internally buffer each individual bus transactions on one side before performing the equivalent transaction on the other side. As shown in Figure 9, a transducer model generated by refinement contains corresponding state machines for each direction of each channel crossing the transducer [13]. Note that each state machine contains its own local buffer, i.e. buffers are not shared, avoiding potential deadlocks and looks up the address mapping table to route the messages over the two different busses.

5. EXPERIMENTAL RESULTS

In order to demonstrate the feasibility and benefits of our approach in terms of design space exploration for a wide variety of designs, we applied our design flow and refinement tool to the design of four industrial-strength examples: a voice codec (*Vocoder*), a JPEG encoder (*JPEG*), an MP3 decoder (*MP3*) and a baseband platform example (*Baseband*) which is combination of a voice codec and JPEG encoder. Different architectures using Motorola DSP56600 processors (*DSP*), Motorola ColdFire processors (*CF*), ARM processors (*ARM*) and custom hardware units (*HW*, *I/O*, *DCT*, *QN*, *FIL*) were generated and various communication architectures (*DSP* bus, *CF* bus, *AMBA* bus and simple handshake bus) were tested.

Table 1 summarizes the features and parameters of the different design examples we tested. For each example, the target communication architecture, the total number of abstract channels and the total traffic in the design are shown. Target architectures are specified as a list of masters plus slaves for each bus in the system where the bus type is implicitly determined to be the protocol of the primary master on the bus. For example, in the case of the *MP3* design, the ARM processor communicates with dedicated hardware units over its *AMBA* bus whereas the *HW* units communicate with each other through separate handshake busses. For simplicity, routing, address and interrupt assignment decisions are not shown in this table.

Overall model complexities are given in terms of code size using lines of code (LOC) as a metric. Results show significant differences in complexity between input and generated output models due to extra implementation detail added between abstraction levels. To quantify the actual refinement effort, the number of modified lines (*Mod. LOC*) is calculated as the sum of lines inserted and lines deleted whereas code coming from database of predefined communication codes (*DB LOC*) is excluded[1]. We optimistically assume that a person can write 30 lines of correct (tested and debugged) code per day. Thus, manual refinement would require hundreds of man-days for reasonably complex designs. Automatic refinement, on the other hand, completes in the order of seconds. Note that in all cases, architecture modes at the input do not have to be changed, i.e. input models remain the same throughout exploration[2]. Results therefore show that a productivity gain of about 1000 times can be achieved using the presented approach with automatic model refinement.

Figure 10 shows the results of exploration of the design space for the MP3 decoder example. We used four different architectures for the MP3 decoder as shown in Table 1. We measured whole system delay of each architecture and the

[1]Our experimental DB includes models of ARM, ColdFire and Motorola DSP including associated busses. On average, DB models have complexities of 1000 LOC per processor/bus combination.

[2]In both cases, design decisions about the target architecture are assumed to be given, i.e. time required for decision making is the same and hence not considered further.

Table 1: Experimental results for different exploration examples.

Examples		Busses (Masters → Slaves)	Chnls (no.)	Traffic (bytes)	Model (LOC)		DB (LOC)	Mod. (LOC)	Refine. Time	
					Arch	TLM			Tool	Manual
Vocoder	A1	DSP → 2 I/Os	5	264718	13047	14937	969	1037	< 1 s	35 days
	A2	DSP → HW, 2 I/Os	17	296014	13978	16492	971	1869	< 2 s	63 days
JPEG	A1	CF → MEM	1	80676	4425	5623	624	604	< 1 s	20 days
	A2	CF → DCT,MEM	9	82170	5148	7207	1000	1221	< 1 s	41 days
	A3	CF → DCT, DMA DMA → MEM, DCT	12	82324	5258	7228	1000	1670	< 2 s	55 days
	A4	CF → DCT, QN, DMA DMA → MEM, DCT, QN	16	83098	5474	8733	1386	2123	< 3 s	71 days
MP3	A1	ARM → I/O	6	21390	28140	33284	4139	1171	< 2 s	40 days
	A2	ARM → I/O, FIL1, FIL2	50	10006	29030	35716	4139	3489	< 5 s	117 days
	A3	ARM → FIL1, FIL2 I/O ↔ FIL1 I/O ↔ FIL2	50	10006	29030	36090	4525	3475	< 5 s	116 days
	A4	ARM → FIL1, FIL2 I/O ↔ Q1 ↔ FIL1 I/O ↔ Q2 ↔ FIL2	52	10006	29160	36754	4525	4035	< 6 s	135 days
Baseband	A1	DSP → BIO,SIO,HW,TX CF → DMA,TX,BR DMA → MEM,BR BR → DCT	29	3298416	19072	24882	2051	4343	< 8 s	144 days

simulation time of each model. As shown in Figure 10, as the number of system components increases with each architecture, the overall performance of the system is improved. In addition, TLMs are as accurate as pin-accurate models (PAMs) but improve simulation speed by around 1000 times compared to PAMs. Given the design decisions made by the user, it took less than 1 hour to obtain 4 different communication models from an executable specification model by architecture exploration [12] and communication design.

6. CONCLUSION

In this paper, we presented an approach for generation of TLMs for SoC communication designs from a partitioned virtual architecture model of a system. A corresponding transaction-level refinement tool has been developed and integrated into our SoC design environment.

Using industrial-strength examples, the feasibility and benefits of the approach have been demonstrated. Automating the tedious and error-prone process of refining a high-level, abstract description of the design into an actual implementation results in significant gains in designer productivity, thus enabling rapid, early exploration of the communication design space. In the future, we plan to integrate IP components with fixed, pre-defined communication protocol interfaces and add algorithms for automated design making for optimization.

7. REFERENCES

[1] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proc. of DATE'03*.

[2] CoWare Platform Architect. Available at http://www.coware.com/products/platformarchitect.php.

[3] ARM MaxSim Tools. Available at http://www.arm.com/products/DevTools/MaxSim.html.

[4] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Mar. 2002.

[5] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Jan. 2000.

[6] L. Cai, and D. D. Gajski. Transaction Level Modeling: An Overview. In *Proc. of CODES+ISSS'03*.

[7] T. Y Yen, and W. Wolf. Communication synthesis for distributed embedded systems. In *Proc. of ICCAD'95*.

[8] R. B Ortega, and G. Borriello. Communication synthesis for distributed embedded systems. In *Proc. of ICCAD'98*.

[9] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proc. DAC'04*.

[10] K. Lahiri, A. Raghunathan, and S. Dey. Efficient exploration of the SoC communication architecture design space. In *Proc. ICCAD'00*.

[11] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proc. DAC'01*.

[12] J. Peng. *System-Level Automatic Model Refinement*. PhD thesis, University of California, Irvine, Information and Computer Science, April 2004.

[13] A. Gerstlauer, G. Schirner, D. Shin, and J. Peng. Necessary and sufficient functionality and parameters for SoC Communication. CECS, Univ. of California, Irvine, Tech. Rep. CECS-TR-06-1, May 2006.

[14] G. Schirner, and R. Dömer. Quantitative analysis of transaction level models for the AMBA bus. In *Proc. of DATE'06*.

[15] A. Wieferink, R. Leupers, G. Ascheid, H. Meyer, T. Michiels, A. Nohl and T. Kogel. Retargetable generation of TLM bus interfaces for MP-SoC platforms. In *Proc. of CODES+ISSS'05*.

[16] H. Yu, A. Gerstlauer, and D. D. Gajski. RTOS scheduling in transaction level models. In *Proc. of ISSS'03*.