Network Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski

Technical Report CECS-04-15
June 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, gajski}@cecs.uci.edu

Network Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski

Technical Report CECS-04-15
June 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, gajski}@cecs.uci.edu

Abstract

Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs. In this report, we propose automatic generation of communication topology from partitioned, scheduled architecture model where system components communicate through message passing channels. Automatic model refinement for network topology enables rapid design space exploration in order to achieve the required productivity gains. The experimental results show the benefits of our methodology and demonstrate the effectiveness of our automatic model generation for communication design.

Contents

1. Introduction **1**

2. Related Works **2**

3. Refinement-based Network Synthesis **3**
3.1. Input Model: Architecture Model . . . 3
3.2. Output Model: Communication Link Model . . . 4
3.3. Network Protocol Library . . . 5
3.4. Design Decisions . . . 5
3.4.1 Bus Allocation and Protocol Selection . . . 5
3.4.2 Communication Element Selection . . . 6
3.4.3 Connectivity Definition . . . 6
3.4.4 Channel Mapping . . . 6
3.4.5 Byte Layout of Memory . . . 6

4. Tasks for Network Refinement **6**
4.1. Channel Grouping . . . 7
4.2. IP/Memory Link Model Generation and Insertion . . . 8
4.3. Protocol Stack Generation and Insertion . . . 9
4.3.1 Presentation Layer . . . 9
4.3.2 Protocol Stack Insertion . . . 9
4.4. Communication Element Synthesis and Insertion . . . 10
4.4.1 Bridge . . . 10
4.4.2 Memory Interface Controller . . . 10
4.4.3 Communication Element Insertion . . . 11
4.5. Logical Link Channel Generation . . . 11

5. Experimental Results **12**

6. Conclusions **14**

List of Figures

1 Refinement-based communication design methodology. 1
2 Communication synthesis flow. 2
3 Communication mechanisms in abstract channels. 4
4 Input model: architecture model. 4
5 Output model: communication link model. 4
6 A screenshot of bus allocation and communication element selection. 5
7 A screenshot of bus mapping. 6
8 Connectivity definition for Figure 4. 7
9 An example for channel grouping. 7
10 Conflict graph for components for Figure 9. 8
11 Merged graphs of the conflict graphs in Figure 10. 8
12 Design after channel grouping for Figure 9. 8
13 Memory model in presentation layer. 9
14 Presentation layer adapter channel for general bus access. 9
15 Presentation layer adapter channel for memory access. 10
16 Bridge behavior in link Model. 11
17 Memory interface behavior in link Model. 11

Network Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs. In this report, we propose automatic generation of communication topology from partitioned, scheduled architecture model where system components communicate through message passing channels. Automatic model refinement for network topology enables rapid design space exploration in order to achieve the required productivity gains. The experimental results show the benefits of our methodology and demonstrate the effectiveness of our automatic model generation for communication design.

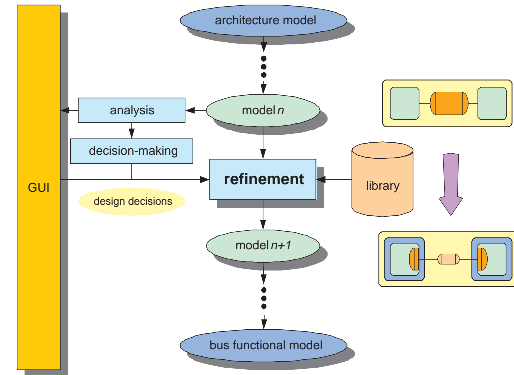

Figure 1. Refinement-based communication design methodology.

1. Introduction

With the ever increasing complexity of system level designs and the pressure of the time-to-market in the design of System-on-Chip (SoC), communication between components is becoming more and more important. Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs.

We propose refinement-based communication design methodology which is a set of models and transformations between models that subdivide the design flow into smaller, manageable steps as shown in Figure 1. With each step, a new model of the design is generated, where a model is a description of design at certain level of abstraction, usually captured in system level design languages. The abstraction level of each model is defined by the amount of implementation detail in terms of structure or order.

In each of tasks, users can make design decisions manually by using an interactive graphical user interface (GUI), for example, while transformations from one model into another can be accomplished automatically by refinement rules or model guidelines. After each refinement step in the synthesis flow, a corresponding model of system is generated, which means that design decisions made in each design task are reflected in the generated models.

Finally, metrics estimation, designers have to simulate generated model to verify the functionality and to estimate design metrics. In general, the design metrics are not satisfactory in the first trial. Therefore, many iterations of these tasks may be needed for each design step.

Figure 2 shows the communication synthesis flow [Ger03] which is divided into two tasks: *network synthesis* and *communication link synthesis*. During the network synthesis, the topology of communication architecture is defined and abstract message passing channels between system components are mapped into communication between adjacent communication stations (communicating system components, e.g. processing elements, communication elements) of the system architecture. The network topology of communication stations connected by logical link channels is defined, bridges and other communication elements are allocated as necessary, abstract message passing channels are routed over sets of logical link channels. The result of the network synthesis step is a refined communication link model of the system. The communication link model represents the topology of communication architecture where components and

additional communication stations communicate with logical link channels.

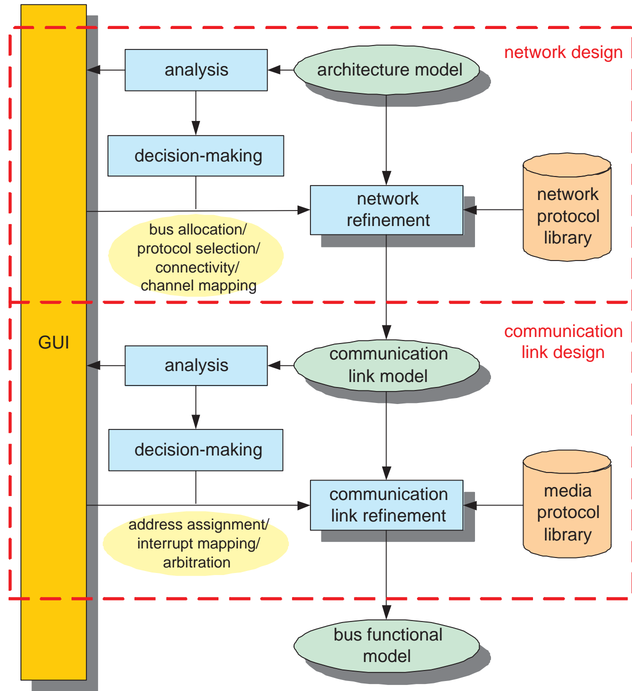

Figure 2. Communication synthesis flow.

There have been some reasons that we have chosen communication link model as an intermediate model in communication synthesis flow [Ger03]. First, designer would like to see the topology of communication architecture and estimate the performance of a communication architecture at early stage of communication synthesis. Secondly, the simulation speed of communication link model is almost same as that of architecture model according to experimental result in [Ger03]. Finally, communication delay of communication link model is not as accurate as that of bus functional model but the accuracy of communication delay in communication link model can be improved by efficient and accurate estimation tools.

Communication link synthesis is followed by network synthesis. Logical links channels between adjacent stations are then grouped and implemented over an actual communication medium (e.g. system busses). During communication link synthesis, each group of logical link channels can be grouped, and be implemented separately onto a communication medium with associated protocol. The parameters such as addresses and interrupts for synchronization are assigned to each logical link channel.

As a result of the communication synthesis process, a bus functional model of a system is generated. The bus functional model is a fully structural model where components 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 bus functional model are then synthesized into targeted hardware or software implementations.

In this report, we will take closer look at network synthesis. The rest of the report is organized as follows: Section 2 gives an overview of related works. In Section 3, we will show the network synthesis flow. Section 4 will present tasks of network refinement followed by the experimental results in Section 5 and finally we will conclude the report with a summary.

2. Related Works

Yen and Wolf [YW95] introduced an iterative approach to generate communication architecture. From initial solution, they find the possible reallocation of a process to another processing element. They choose either a process reallocation or a communication reallocation according to sensitivity analysis during each iteration. When no reallocation remains feasible, new bus is added into the design, in addition to new processing element. These steps are repeated until no reallocation is possible. In this way, they can address the problem of heterogeneous processors connected via arbitrary bus topologies. However they only assume an abstract protocol based on processor priorities.

Gasteier and Glesner [GG96] [GMG98] presented an approach to automatic generation of communication topologies on system-level. Given a set of processes communicating via abstract send and receive functions and detailed information about the communication requirements of each process, they first perform a clustering of data transfers. This results in groups of transfers suited to share a common bus. For each of these clusters, they execute a bus generation algorithm which schedules bus accesses in order to minimize the total communication costs. Other than previous approaches, they infer RAM, if necessary, and consider data-dependencies as well as periodic execution of processes.

Ortega and Borriello [OB98] introduced a communication synthesis algorithm focusing on implementing the communication links between communicating processes in distributed embedded systems. They started from a given mapping of processes to execution units along with a mapping of port connections to busses. They generated the communication patterns and a real-time operating system for each processor. In their approach, an analysis of different communication structures and process mapping are not performed automatically.

Lahiri et al. [LRD00] automatically mapped the various communications between system components onto a target communication architecture template and configures the

communication protocols of each channel in the architecture in order to optimize the system performance by taking the bus conflict into account.

However, most of research work has been on automatic decision-making on communication topology of system architecture. There has been little attention paid to automatic generation of network topology of communication architecture from the partitioned, scheduled architecture model.

3. Refinement-based Network Synthesis

Within our SoC communication design framework, the network synthesis implements end-to-end communication semantics between system components which is mapped into point-to-point communication between communication stations of network architecture [Ger03]. The result of network synthesis is a refined communication link model of the system. In the communication link model, system components and other communication stations communicate via logical link channels that carry streams of bytes between directly connected components.

The network synthesis task consists of implementations for presentation, session, transport, and network layers [Ger03]. The presentation layer is responsible for data formatting. It converts abstract data types in the application to blocks of ordered types as defined by the canonical byte layout requirements of the lower layers. For example, the presentation layer takes care of component-specific data type conversions and endians (byte order) issues.

The session layer implements end-to-end synchronization to provide synchronous communication as required between system components in the application. Furthermore, it is responsible for multiplexing messages of different channels into a number of end-to-end sequential message streams.

The transport layer implements end-to-end flow control and error correction to guarantee reliable transmission. In standard bus-based SoC design, stations and links are reliable and then transport layer need not be implemented.

Finally, the network layer is responsible for routing and multiplexing of end-to-end paths over individual point-to-point logical links. As part of the network layer, additional communication stations are introduced as necessary, e.g. to bridge two different bus systems. The communication stations split the system of connected system components in architecture model into several bus subsystems. Assuming reliable stations and logical links, routing SoCs is usually done statically, i.e. all packets of a channel take the same fixed, pre-determined path through the system.

Network refinement tool takes three inputs: input architecture model, design decisions and network library. With these inputs, the network refinement tool produces an output model that reflects the topology of communication architecture of the system. In the output model, the top level of the design consists of system components and logical link channels between communication stations. The logical link channes themselves are refined to their bus implementation during communication link synthesis.

3.1. Input Model: Architecture Model

The architecture model is the starting point for communication design and is the result of the architecture exploration. In the architecture model, system components communicate via message-passing channels. The communication design process gradually implements these channels and generates a new model for each layer of communication functionality inserted.

The architecture model follows certain pre-specified semantics. 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 synthesis. Inter-component communication is end-to-end and takes place through abstract channels that support *send* and *receive* methods.

For each variable communicated between system components, the model contains corresponding typed message-passing channels. Communication between components can be modeled via three schemes as shown in Figure 3. In the case of two way blocking communication as shown in Figure 3, both the sender and receiver must be blocked until the transaction has completed. This mechanism is modeled using events and blocking wait statements. As we can see, the sender writes the data on a shared variable in the channel and follows up by notifying the receiver. The receiver can not read the data until it gets the sender's notification. This guarantees the safety of the transaction. The *ack* event guarantees that the sender cannot rewrite on the channel until the previous transaction has completed. Such a mechanism is deterministic.

In Figure 3, the one way blocking mechanism is used that ensures that the receiver cannot read the data until it is written by the sender. However, there is no way to stop the sender from re-writing that data in a subsequent iteration. The non-blocking mechanism shown in Figure 3 is completely non-deterministic. These mechanisms are typically used in real time systems where a time out strategy is employed. In the thesis, we will look only at refinement of two way blocking communication because it is used for unbuffered data transfer and can be implemented directly over standard bus-based communication protocols. The other two mechanisms can be implemented easily once we have support for two way blocking communication [Pen04].

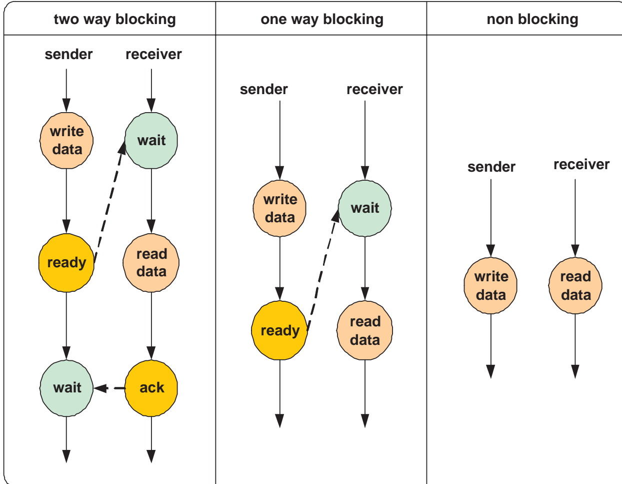

Figure 3. Communication mechanisms in abstract channels.

For example, the architecture model after architecture exploration is shown in Figure 4. During the architecture exploration, the application has been mapped onto a typical system architecture consisting of a processor (*DSP*), a custom hardware (*HW1*), an FPGA (*HW2*) and a system memory (*MEM*). Inside DSP, tasks are dynamically scheduled under the control of an operating system model [GYG03] [YGG03] that sits in an additional operating system shell of the processor (*DSP_OS*). If a processor needs no operating system, the operating system shell is empty.

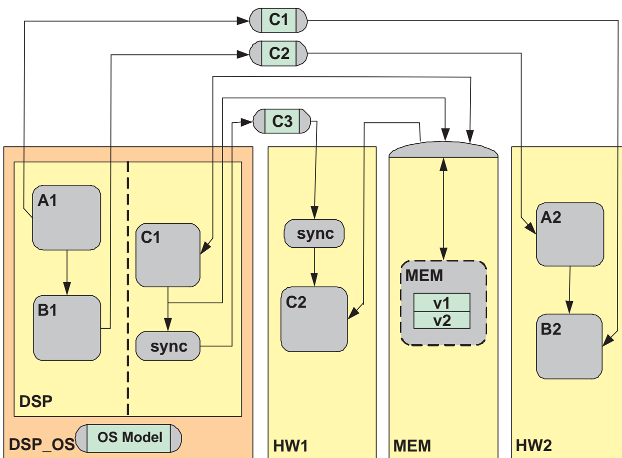

Figure 4. Input model: architecture model.

In the architecture model, the shared memory component is modeled as a special behavior with interfaces. The memory behavior encapsulates all variables mapped into the shared memory component. At its interface, the memory behavior provides two methods for each variable to read and write the value of the variable from/to memory.

3.2. Output Model: Communication Link Model

The communication link model is an intermediate model for communication design and is the result of the network synthesis and reflects communication topology of the communication architecture. Also, communication link model serves as specification of communication link synthesis which is followed by network synthesis. In the communication link model, we can see system components communicating via logical link channels which still implements message passing semantics.

For each variable communicated between components, the implementations of upper layers of protocol stack such as presentation layer, session layer, transport layer and network layer are inlined into the corresponding system components. In the communication link model, end-to-end application channels have been replaced with point-to-point logical link channels between system components that will later be physically directly connected through bus wires during communication link synthesis. In the communication link model, communication elements such as bridges, transducers are inserted from network protocol library and synthesized to perform protocol translation between two different busses.

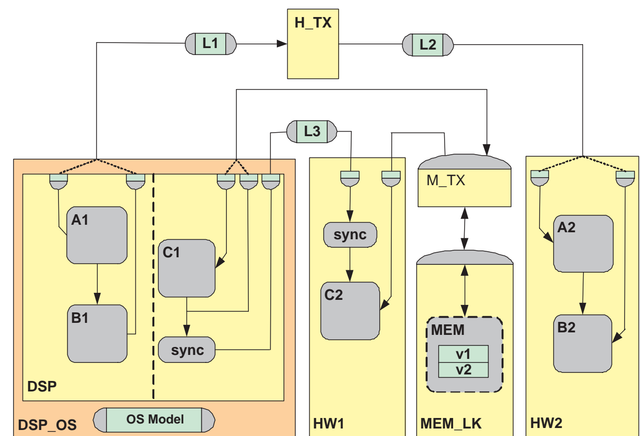

Figure 5. Output model: communication link model.

For example, the communication link model after network synthesis is shown in Figure 5. During the network synthesis, the application channels (*c1*, *c2*, *c3* and *MEM*) are inlined into the behavior of the corresponding system components and also the implementations of presentation layer are inserted and connected with the application layer adapter channels in the model. The application channels (*c1* and *c2*) which are accessed sequentially in *DSP* and *HW2*,

have been grouped but mapped onto two different busses (*DSPBus* and *HBus*). A bridge (*H_TX*) is introduced to perform protocol translation between *DSPBus* and *HBus*. The communication between *DSP* and *HW2* is routed over logical link channels, *L1* and *L2* via *H_TX*. The communication by message passing channel *c1* between *DSP* and *HW1* is mapped over *L3*.

Also a system memory (*MEM*) has its own interface protocol (*MBus*) which has been connected to the *DSPBus* and therefore, a transducer (*M_TX*) for protocol translation between them is necessary. The transducer is modeled by a special behavior with interfaces. At its interface, the transducer behavior provides two methods (*send/receive)* to send and read the value of variable to/from memory.

But memory component model has to be refined down to an accurate representation of the byte layout. All variables stored inside memory are replaced with and grouped into a single array of bytes. The memory component is still modeled as a special behavior with an interface which provides two methods (*read/write*) to access each variable mapped onto it with the offset of the variable in the memory.

3.3. Network Protocol Library

The network protocol library is a set of channels that represent the protocols of system busses and behaviors that represent communication elements on the system busses. The channels for bus protocols contain attributes of the busses including the name of associated protocols, address width, data width, etc. The actual implementation of bus protocols is defined in media protocol library which will be used in communication link synthesis. The behavior for communication elements contains also the attributes of the communication elements including name, type of the communication elements and bus protocols associated.

3.4. Design Decisions

The refinement engine works on directions given to it by design decisions. The design process can either be automated or interactive as per user's methodology. During network synthesis, design decisions include *allocation of system busses and protocol selection*, *selection of communication elements*, *the connectivity definition between components and busses*, *the mapping of abstract communication to busses*, and *byte layout of system memories*. Based on these decisions, the refinement engine maps the application channels onto logical links and imports the templates of the required communication elements from the network protocol library and synthesizes the implementations of the communication elements and finally generates communication link model.

For the purpose of our implementation, we annotated architecture model with the set of design decisions. The refinement tool then detects and parses these annotations to perform the requisite model transformations. Based on these decisions, the refinement engine imports the required communication elements from the network protocol library and produces a communication link model that reflects communication topology of the bus architecture of the system. In the communication link model, the top level of the design consists of system components and logical link channels to connect system components.

3.4.1 Bus Allocation and Protocol Selection

In bus allocation step, the number of system busses is determined and the associated protocol to each bus need to be decided. For bus allocation and protocol selection, the communication traffics between components, the bus bandwidth and concurrency of channel accesses in the components need to be considered. However, for components with fixed, predefined interface protocols, the bus allocation and protocol selection are automatically performed on the fly from network protocol library. For example, ARM processor is included in a design, then AMBA bus interface should be allocated during bus allocation and protocol selection.

Figure 6 show the screenshot of the bus allocation and communication element selection in our SoC design environment. In the screenshot, *Busses* and *CEs* in a *Bus Allocation Window* are for allocation of busses and communication elements. During the bus allocation, GUI shows parameterizable attributes such as bit width, bandwidth, and baud rate of the selected bus.

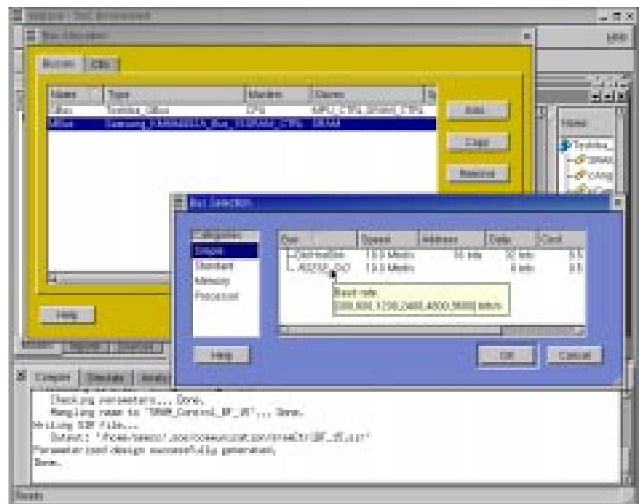

Figure 6. A screenshot of bus allocation and communication element selection.

3.4.2 Communication Element Selection

As part of the network layer, additional communication elements are introduced as necessary, e.g. to create and bridge subnets using communication elements, splitting the system of directly connected components with message passing channels in the architecture model into several bus groups.

The communication elements include bridges connecting two different busses for example, AMBA bus and PCI bus. Also, for a shared memory access, interface protocol of the memory is usually not compatible to the system bus, then the memory interface controller need to be allocated.

During network refinement, the implementation of the allocated communication elements is synthesized or taken out of network protocol library. In the SoC design environment, the selection of communication elements is implemented in *Bus Allocation Window*.

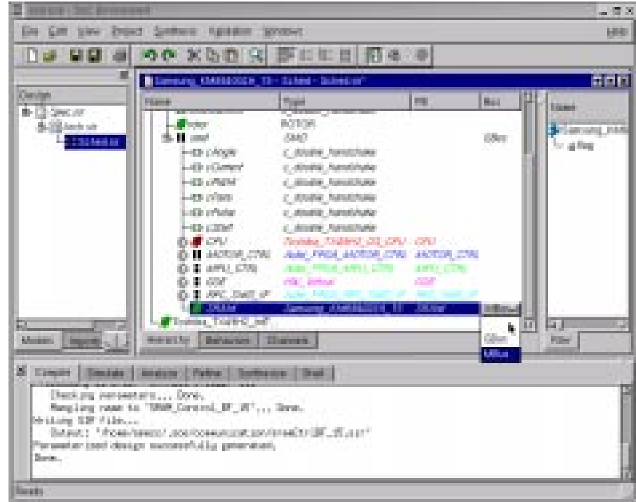

Figure 7. A screenshot of bus mapping.

3.4.3 Connectivity Definition

So far, we have allocated processing elements and storage elements which have refined down to the component structure of a system architecture through architecture exploration. Also, system busses and communication elements through bus allocation and communication element selection are allocated. These system components need to be connected to each other on the system busses in order to transfer data to each other. Therefore, how system components are connected to each other on the allocated system busses need to be defined, in order words, topology of communication architecture (connectivity) need to be defined. Then, messages will be routed over the topology of communication architecture to perform transactions between components.

The connectivity definition is implemented graphically or tabular form as shown in Figure 7. In the screenshot, *SRAM* is assigned to *MBus* protocol, which becomes the interface protocol of *SRAM*.

3.4.4 Channel Mapping

Now, the topology of communication architecture has been defined, and then message passing channels on top of a design in architecture model must be mapped to the allocated system busses and communication elements. For example, channels *c1* and *c2* in Figure 4 will be mapped to *DSPBus* and *HBus* and communicate through communication element *H_TX*. This channel mapping is decided by users with the help of GUI. The mapping information is annotated into its corresponding channel in a design.

3.4.5 Byte Layout of Memory

As part of presentation layer implementation, the shared memory component model has been refined down to an accurate representation of the byte layout. All variables stored inside memory are replaced with and grouped into a single array of bytes. Layout and addressing of variables inside the memory need to be defined based on the lifetime analysis of each variable and the alignment of the chosen target memory component. For example, if two variables have non-overlapping lifetimes, they can be stored in the same memory locations, since the same memory space can be reused for the two arrays [PDN99].

4. Tasks for Network Refinement

Network refinement tool refines the input, partitioned, scheduled architecture model into a communication link model that reflects communication topology of a system. The refinement process can be divided into five steps, namely, *channel grouping*, *IP/Memory link model generation and insertion communication element synthesis and insertion*, *protocol stack generation and insertion*, and *link layer generation and instantiation*, each can be further divided into sub-steps.

In this section, we illustrate the network refinement process through a simple yet typical example as shown in Figure 4. We assume that three busses (*DSPBus* for *DSP*, *HBus* for *HW2* and *MBus* for *MEM*) are allocated and the connectivity is defined as shown in Figure 8.

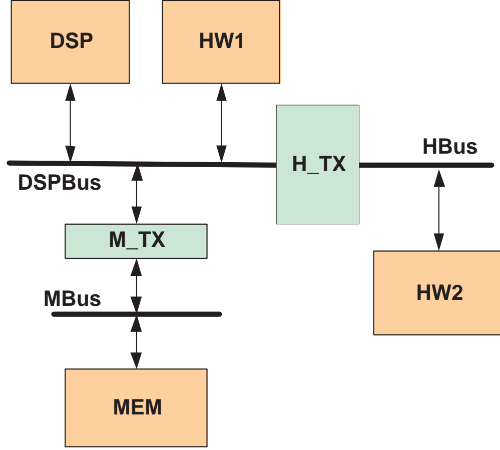

Figure 8. Connectivity definition for Figure 4.

4.1. Channel Grouping

Once communication topology for a system is defined, abstract transactions by message passing channels between components in the architecture model need to be mapped onto the communication topology of the system. If all transactions between two components are sequential, therefore, no concurrent communication can be multiplexed over logical links. In other words, sequential transactions are merged over single streams as much as possible in order to reduce number of logical link channels in the system.

Let us look at the following example in Figure 9. Three double handshake channels are used for the message passing between three components and are mapped into one bus. On *PE1*, *A1* is followed by a parallel composition of *B1* and *C1* while on *PE2*, *A2* is followed by *B2* and *C2* is executed on *PE3*. Channel *cB* and *cC* can not be shared, because the execution order between *B1* and *C1* is not known in advance. If we shared *cB* and *cC*, then *B1* would potentially receive data from *C1* which was intended for *C1*. However, we can safely share *cA* and *cB* because we know that *A1* is always executed first.

With this observation, we can turn the channel grouping problem into a *graph coloring problem* [CLR90] as follows:

Given: Conflict graph $G(V,E)$ for each component, where the vertices V correspond to channels which are mapped to a same bus and the edges E represented concurrency between channels.

Determine: Minimum number of channels concurrent

The graph coloring problem is NP-complete [CLR90]. Heuristic graph coloring algorithms can be used here. First we can build a conflict graph for each component based on the sequentiality and parallelism analysis on the channels.

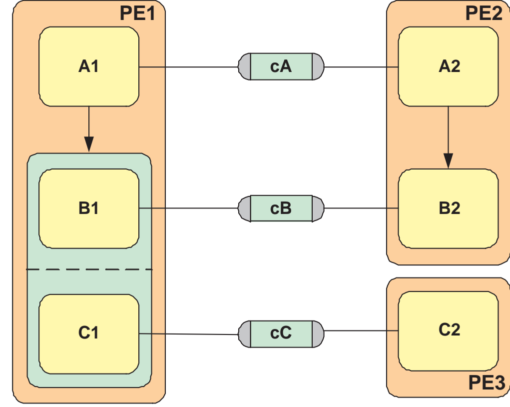

Figure 9. An example for channel grouping.

We define that channel *c1* and channel *c1* are *conflict* to each other, if channel access of *c1* executes in parallel with the that of *c2* in a component.

Algorithm 1 shows how to build conflict graph for a PE from a design. The input to Algorithm 1 is the internal representation from the whole design IR_{Design}. First, channels C that are connected to each component B_{PE} are the vertex of the conflict graph for each component (line 3 – line 5). In order to find whether or not accesses for the channels are executed concurrently, we look into all child behaviors inside the component and see that the behavioral compositions of the child behaviors are parallel or sequential (line 7). If the channels are accessed in parallel in each component, the edge between the channels are inserted (line 8).

Algorithm 1. BuildConflict (IR_{Design})

```
1:  for all B_PE ∈ IR_Design do
2:      G_PE(V,E) = {}
3:      for all C ∈ IR_Design do
4:          if IsPort (C, B_PE) then
5:              AddVertex (G_PE(V,E), v_C)
6:              for all v ∈ G_PE(V,E) do
7:                  if IsExecutedInParallel (C, v) then
8:                      AddEdge (G_PE(V,E), {v_C, v})
9:                  end if
10:             end for
11:         end if
12:     end for
13: end for
```

The conflict graphs of our simple example in Figure 9 are shown in Figure 10. The channels *cA*, *cB* and *cC* are connected to ports of the *PE1* and the *B1* for the channel *cA* and *C1* for the channel *cC* are executed in parallel on *PE1*. Thus, vertex *cB* and *cC* have an edge in the conflict graph

of *PE1*.

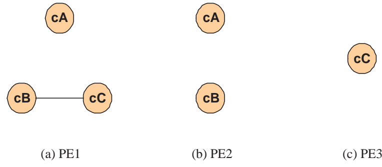

Figure 10. Conflict graph for components for Figure 9.

Now, we have conflict graphs for each component and then we have to merge two conflict graphs of two components in order to get a conflict graph where graph coloring is performed. Algorithm 2 shows the algorithm to merge the conflict graphs of two components. First, merging step starts a common graph $G_{PE12}(V,E)$ with no vertex and edge (line 2). The common set of vertices of two conflict graphs of two components are added into the common graph (line 3 – line 4). Then, if either of the conflict graphs for each component has an edge between newly added vertex v_{PE1} or v_{PE2} and vertices v in the common graph, the edge v_{PE1}, v or v_{PE1}, v is inserted (line 5 – line 7).

Algorithm 2. MergeGraph ($G_{PE1}(V,E)$, $G_{PE2}(V,E)$)

```
1:  for all v_PE1 ∈ G_PE1(V,E), v_PE2 ∈ G_PE2(V,E) do
2:    G_PE12(V,E) = {}
3:    if v_PE1 == v_PE2 then
4:      AddVertex (G_PE(V,E), v_PE1)
5:      for all v ∈ G_PE12(V,E) do
6:        if ExistEdge (G_PE1(V,E), {v_PE1, v}) || Exist-
          Edge (G_PE2(V,E), {v_PE2, v}) then
7:          AddEdge (G_PE12(V,E), {v_PE1, v});
8:        end if
9:      end for
10:   end if
11: end for
```

After merging the conflict graphs of two components, the result graphs are shown in Figure 11

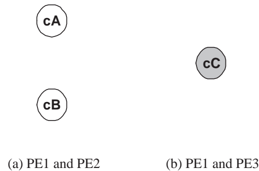

Figure 11. Merged graphs of the conflict graphs in Figure 10.

Finally, we have to color each merged graphs in order to get the minimum number of channels between two components. We can use any heuristic algorithm for graph coloring [CLR90]. In Figure 11, we can get one color in each merged graph. Therefore, the result model after channel grouping for Figure 9 is shown in Figure 12. The channel *cA* and *cB* between *PE1* and *PE2* will be grouped and shared in the channel *cAB* in Figure 11.

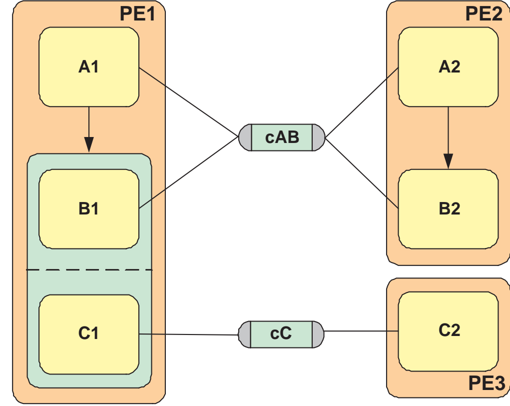

Figure 12. Design after channel grouping for Figure 9.

The two channels *c1* and *c2* as shown in Figure 4 can be merged because accesses to the channels in two component *DSP* and component *HW2* are sequential. In the communication link model as shown in Figure 5, the two logical link channels *L1* and *L2* reflects sharing of two channels (*c1* and *c2*).

4.2. IP/Memory Link Model Generation and Insertion

During network synthesis, a purely behavioral IP component has been replaced with a model that encapsulates a structural model of the component in a wrapper that implements all layers of communication with the IP. Since the IP's communication protocol are pre-defined and fixed, its communication not be designed arbitrarily and the wrapper provides the necessary functionality to be gradually inserted into the IP's communication partners as design progresses.

For a memory component, all variables stored inside memory are replaced with and grouped into a single array of bytes (line 4). Again the behavior model for memory component implement *read* and *write* interface so that the components accessing the memory can use the interface method calls (*IMemLink* in line 2, and line 6 – line 11).

For example, the memory component (*MEM*) in the architecture model is replaced with its link model (*MEM_LK*) in the communication link model as shown in Figure 5.

Figure 13. Memory model in presentation layer.

```
1 #define MEM_SIZE 1024
2 behavior Memory(void) implements IMemLink
3 {
4     char mem[MEM_SIZE];
5     void main(void) { }
6     void read(unsigned long int offset, void *
          data, unsigned long int len) {
7         memcpy(data, mem + offset, len);
8     }
9     void write(unsigned long int offset, void *
          data, unsigned long int len) {
10        memcpy(mem + offset, data, len);
11    }
12 };
```

Figure 14. Presentation layer adapter channel for general bus access.

```
1 channel cIntPresent(INetwork network) implements
      IIntPresent
2 {
3   void receive(int* val) {
4     uint32_t data;
5     network.receive(&data, sizeof(data));
6     *val = ntohl(data);
7   }
8   void send(int val) {
9     uint32_t data;
10    data = htonl(val);
11    network.send(&data, sizeof(data));
12  }
13 };
```

4.3. Protocol Stack Generation and Insertion

The network synthesis implements presentation, session, transport, and network layers. In SoC communication, the session, transport and network layer are mostly empty. Thus, we will insert the implementation of presentation layer into each component.

4.3.1 Presentation Layer

The presentation layer is responsible for data formatting. It converts abstract data types in the application to blocks of ordered types as defined by the canonical byte layout requirements of the lower layers as shown in Figure 14. If the bit width of a variable is smaller than the bus width, then the variable can be assigned, for example, to the lower bits of the bus. If the bit width of a variable is greater than the bus width, then the words have to be transfered in different bus cycles, for example, in a sequence of bus transfers. The sequence can be written using little endian or big endian assignment. Little endian assigns the least significant bits (LSB) to the lowest address. In contrast, big endian assign the most significant bits (MSB) to the lowest address. For example, the presentation layer takes care of component-specific data type conversions and endians (byte order) issues (line 6 and line 10)

In the communication link model, the model for components contains presentation layer implementations in the form of adapter channels that provide the service (interface) of the presentation layer to the application on one side (*IPresent*), while connecting and calling network layer methods on the other side (*INetwork network* in line 1). The presentation layer performs data formatting for every message data type found in application (integer to void pointer type conversion in line 5 and line 11). Therefore, each application layer adapter channel can be connected to corresponding presentation layer adapter one by one. Since the presentation layer becomes a part of the application, its adapter channels are instantiated inside the application of each component.

The presentation layer inside the component accessing a global, shared memory are responsible for converting variables in application into size and offset for shared memory accesses as shown in Figure 15. For example, The first arguments of interface method calls (line 5, line 11, line 15, line 21) to the memory behavior are aligned based on the size of the variables. In case of array accesses in the application, the presentation layer for the memory takes the indices of arrays as arguments of the interface methods (line 13, line 18). The indices will be added to base address of the array in the memory when the presentation layer accesses the memory behavior through interface method calls (line 15, line 21).

4.3.2 Protocol Stack Insertion

The protocol stack insertion process is shown in Algorithm 3. The input to Algorithm 3 is the internal representation for the whole design IR_{Design}. Each component behavior B_{PE} inside the design is checked if it is software component or hardware component. If the component is software, the presentation layers ($C_{present}$) as shown in Figure 14 and Figure 15 if the component accesses memory, are created with the corresponding operating system (OS) interface port (P_{os}) inside its application shell ($B_{PE_{app}}$) (line line 4 – line 5). Also OS adapter channel (C_{os}) will be instantiated with the link layer channel interface port (P_{link}) inside the OS shell ($B_{PE_{OS}}$) (line 3, line 6). In case of the presentation layer for memory in PE, the memory interface port (P_{mem}) will re-used to create presentation layer for memory (C_{memory}) inside its application shell ($B_{PE_{app}}$) (line 7 – line 10). In the same way, the presentation layers for the hard-

Figure 15. Presentation layer adapter channel for memory access.

```
1  channel cPresentMem(IMemLink shm) implements
       IMemApp
2  {
3      int read_A(void) {
4          int val;
5          shm.read(0ull, &val, sizeof (int));
6          return val;
7      }
8      void write_A(int data) {
9          int val;
10         val = data;
11         shm.write(0ull, &val, sizeof (int));
12     }
13     char* read_B(unsigned int idx) {
14         char* val;
15         shm.read(4ull+idx, &val, sizeof (char));
16         return val;
17     }
18     void write_B(char* data, unsigned int idx) {
19         char* val;
20         val = data;
21         shm.write(4ull+idx, &val, sizeof (char));
22     }
23 };
```

ware component can be created inside the B_{PE} (line 11 – line 18).

For example, DSP component (*DSP*) is connected to five channels including memory components in the architecture model. The five presentation layers are generated and inlined into the DSP component and connected with applications in the communication link model as shown in Figure 5.

4.4. Communication Element Synthesis and Insertion

As part of network synthesis, communication elements might have to be inserted into the system architecture. The communication elements that translate between incompatible bus protocols will act as bridges connecting two busses or as bus interfaces for components with fixed, predefined protocols. Like the other components, the behaviors of the communication elements are instantiated and added to the set of concurrent, non-terminating components at the top level of the design.

If a communication element is allocated from the protocol library, its functionality is not implemented yet. We need to fill functionality for the communication element by synthesis.

Algorithm 3. InsertStack (IR_{Design})

1: **for all** $B_{PE} \in IR_{Design}$ **do**
2:   **if** $B_{PE} == SW$ **then**
3:     CreatePort ($B_{PE_{OS}}$, P_{link})
4:     CreatePort ($B_{PE_{app}}$, P_{os})
5:     CreateInstance ($B_{PE_{app}}$, $C_{present}$, P_{os})
6:     CreateInstance ($B_{PE_{OS}}$, C_{os}, P_{link})
7:     **if** hasMemoryPort (B_{PE}) **then**
8:       P_{mem} = FindMemoryPort (B_{PE})
9:       CreateInstance ($B_{PE_{app}}$, C_{memory}, P_{mem})
10:     **end if**
11:   **else if** $B_{PE} == HW$ **then**
12:     CreatePort (B_{PE}, P_{link})
13:     CreateInstance (B_{PE}, $C_{present}$, P_{link})
14:     **if** hasMemoryPort (B_{PE}) **then**
15:       P_{mem} = FindMemoryPort (B_{PE})
16:       CreateInstance (B_{PE}, C_{memory}, P_{mem})
17:     **end if**
18:   **end if**
19: **end for**

4.4.1 Bridge

Figure 16 shows the behavioral model of a bridge used in communication link model. The bridge has the two interface ports for two different busses (*bus1* and *bus2*). In the most case, the bridge will listen on both sides simultaneously in order to handle transfers dynamically as they come in. Therefore, the bridge consists of two parallel child behaviors *ForwardBhvr* and *BackwardBhvr*, each of which is responsible for transferring data in one direction and the other direction.

The complete messages are received on one side, buffered in the bridge's local memory (*forward[BUF_SIZE]* in line 4, *backward[BUF_SIZE]* in line 14), and sent out on the other side. The variables inside the behaviors will model the buffer which will be further implemented to FIFO queue or memory later.

4.4.2 Memory Interface Controller

Since memory components have their own fixed interface protocol, they might not be directly connected to the system bus. Memory interface controller, therefore, might be inserted into communication link model during network refinement.

The interface of the memory interface controller should be handled differently from bridge because link layer model for memory is implemented by behavior with interfaces as shown in Figure 17. The memory interface controller copies the channel interface methods (*read* and *write* methods) of the link model of memory behavior (*IMemLink* in line 1).

Figure 16. Bridge behavior in link Model.

```
1  #define BUF_SIZE 1024
2  behavior ForwardBhvr (IBus1Link bus1, IBus2Link
       bus2)
3  {
4      char forward[BUF_SIZE];
5      void main (void) {
6          while (true) {
7              bus1.receive(forward, BUF_SIZE);
8              bus2.send(forward, BUF_SIZE);
9          }
10     }
11 };
12 behavior BackwardBhvr (IBus1Link bus1, IBus2Link
       bus2)
13 {
14     char backward[BUF_SIZE];
15     void main (void) {
16         while (true) {
17             bus2.receive(backward, BUF_SIZE);
18             bus1.send(backward, BUF_SIZE);
19         }
20     }
21 };
22 behavior Bridge (IBus1Link bus1, IBus2Link bus2)
23 {
24     ForwardBhvr Forward (bus1, bus2);
25     BackwardBhvr Backward (bus1, bus2);
26     void main (void) {
27         par {
28             Forward.main();
29             Backward.main();
30         }
31     }
32 };
```

Inside interface methods, the memory interface controller invokes the link layer interface methods (line 5, line 8).

Figure 17. Memory interface behavior in link Model.

```
1  behavior MemoryCtrl(IMemLink mem) implements
       IMemLink
2  {
3      void main(void) { }
4      void read(unsigned long int offset, void *
           data, unsigned long int len) {
5          mem.read(offset, data, len);
6      }
7      void write(unsigned long int offset, void *
           data, unsigned long int len) {
8          mem.write(offset, data, len);
9      }
10 };
```

4.4.3 Communication Element Insertion

Algorithm 4 shows how to insert the synthesized communication elements CEs into the internal representation IR_{Design} of a design. First, all busses Bus_{PE1}, Bus_{PE2}, Bus_C connected to components and channels are extracted from connectivity of the design (line 2 – line 4). Basically, if the busses which are mapped to channels and components, are not compatible, we insert the corresponding communication element $CE_{Bus_{PE1},Bus_{PE2}}$, $CE_{Bus_C,Bus_{PE1}}$ (line 5 – line 12).

On the other hand, the memory is connected through its interface instead of channels in input model. Thus we insert the memory interface controller behavior with interface. The interface of the memory controller behavior will be connected to its corresponding component and the original memory will be connected to the interface of the memory controller behavior (line 17 – line 22).

For example, a memory interface controller *M_TX* between *DSPBus* and *MBus* and a bridge *H_TX* between *DSPBus* and *HBus* are synthesized and instantiated in the communication link model as shown in Figure 5.

4.5. Logical Link Channel Generation

In the top level of behavior hierarchy, adjacent components need to be connected with logical link channels which implement two way blocking semantics as described in Section 3.1 because the communication link model has to preserve the original semantics of input architecture model.

The number of logical link channels are determined by channel grouping algorithm as shown in Section 4.1. For the implementation, we reuse application channels in the architecture model which implement two way blocking semantics and instantiate them to connect adjacent components in the communication link model.

Algorithm 4. InsertCEs (IR_{Design}, CEs)

```
1:  for all channel C ∈ IR_Design do
2:    Bus_C = mapped bus of C
3:    Bus_PE1 = mapped bus of the sender of C
4:    Bus_PE2 = mapped bus of the receiver of C
5:    if Bus_C ≠ Bus_PE1 then
6:      FindCE (CEs, CE_{Bus_C,Bus_PE1})
7:      CreateInstance (IR_Design, CE_{Bus_C,Bus_PE1})
8:    end if
9:    if Bus_C ≠ Bus_PE2 then
10:     FindCE (CEs, CE_{Bus_C,Bus_PE2})
11:     CreateInstance (IR_Design, CE_{Bus_C,Bus_PE2})
12:   end if
13: end for
14: for all memory M ∈ IR_Design do
15:   Bus_M = mapped bus of M
16:   Bus_PE = mapped bus of the partner of M
17:   if Bus_M ≠ Bus_PE then
18:     FindCE (CEs, CE_{Bus_M,Bus_PE})
19:     CreateInstance (IR_Design, CE_{Bus_M,Bus_PE}, M)
20:     P_M = FindMemoryPort (B_PE)
21:     ReplacePort (P_M, CE_{Bus_M,Bus_PE})
22:   end if
23: end for
```

For example, three logical link channels (*L1*, *L2* and *L3*) are instantiated and connected with components including an additional communication element (*H_TX*) as shown in Figure 5.

5. Experimental Results

Based on the described methodology and algorithms, we developed a network refinement tool, called *SpecC Network Refinement (scnr)*, which takes architecture model and user decisions for network synthesis and generates communication link model. For experiments, we generated architecture models for different examples, JPEG and GSM Vocoder.

Table 1 shows the characteristics of the partitioned architecture models of these examples. Different architectures using Motorola DSP56600 processor (DSP), MIPS based CPU (CPU) and custom hardware (HW) were generated and various bus architectures were tested. In Table 1, the number of channels represents the number of message passing channels in top level of architecture model and the total traffic refers to the amount of data exchanged between components.

Table 2 shows design decisions which are made during network synthesis. In this table, channel mapping from application channels to link channels is done by automatic network refinement tool which implements the channel grouping as shown in Section 4.1.

To compare against the manual effort of model refinement, we have to measure the quality of the generated models. In order to be able to assess the model quality, we propose a set of quality metrics. Some of them are quantitative but others are difficult to be quantized. These quality metrics are important to compare the automatic refinement with manual refinement. Although the automatic refinement has the advantage of error-proof and being fast, its output may not be as good as the one produced manually by designers. Therefore, we need to define good metrics to evaluate the quality of the refinements.

The first metrics we can use is the code size of the models. In general, each refinement will increase the size of the model because more details are added. The code size can be quantitatively measured by the number of source lines. Therefore, we can use (number of lines of original model / number of lines of refined model) as one of the quality metrics. The smaller the ratio, the better the refinement. In general, more model objects is equivalent to longer code. So the creation of new objects should be avoided as much as possible.

The second metrics we can use is the communication overhead introduced by the refinement. As we explained earlier, the channels accessed sequentially in the behaviors can be merged, which improves the quality of the models. First, the merging process removes redundant channels which cause the code size increase. In addition, merged channels will reduce the number of channels for optimization which users have to decide.

The final quality we can look at is the readability of the refined model. It is difficult to define quantitatively the readability. However, in general, the readability can be improved by keeping the changes as small as possible and by following a simple but self-explanatory naming convention.

Table 3 shows the results of network refinement. We used the Lines of Code (LOC) metric and number of channels reduced during network refinement. The number of channels are reduced based on channel grouping as shown in third and fourth column in Table 2 which turns out to reduce the number of lines of codes significantly in JPEG and Vocoder example. Modified lines of code by automatic refinement is calculated by $modified = inserted - library + deleted$ (lines of code from library need to be subtracted in calculation of modified lines of code) is in fifth column. Even with a very optimistic estimate of 10 LOC per person hour, manual refinement takes several weeks to get communication link models. Automatic refinement, on the other hand, completes in the order of a second. In order to calculate the productivity gain, we assume that design decisions (bus allocation, protocol selection, channel mapping) for network design by a designer take 5 minutes per bus. For example, the productivity gain for *arch1* of JPEG is calcu-

Table 1. Characteristics of the partitioned architecture models.

Examples		Components	Buses	# Channels	Traffic
JPEG	arch1	1 DSP, 1 HW	1 DSP Bus	7	7560
	arch2	1 DSP, 1 DCT IP	1 DSP Bus	2	2160
Vocoder	arch1	1 DSP, 1 HW	1 DSP Bus	12	46944
	arch2	1 DSP, 2 HW	1 DSP Bus	22	56724
	arch3	1 DSP, 3 HW	1 DSP Bus	42	76284
	arch4	2 DSP, 2 HW	1 DSP Bus	29	52160

Table 2. Design decisions for network synthesis.

Examples		Channels	Routing	Medium (master/slave)
JPEG	arch1	all	linkHW	DSP Bus (DSP/HW)
	arch2	all	linkIP	DSP Bus (DSP/IP)
Vocoder	arch1	all	linkHW	DSP Bus (DSP/HW)
	arch2	12 channels	linkHW1	DSP Bus (DSP/(HW1, HW2))
		10 channels	linkHW2	
	arch3	12 channels	linkHW1	DSP Bus (DSP/(HW1, HW2, HW3))
		11 channels	linkHW2	
		19 channels	linkHW3	
	arch4	17 channels	linkDSP1HW1	DSP1 Bus (DSP1, DSP2)/(HW1, HW2)
		12 channels	linkDSP2HW2	

Table 3. Experiment results for network refinement.

Examples		Lines of Code			Automatic refinement	Manual refinement	Productivity gain
		Arch	Link	Modified (inserted (lib)/deleted)			
JPEG	arch1	2940	2969	133 (81 (0)/52)	0.08 s	13.3 hr	160
	arch2	2717	2763	88 (66 (0) /22)	0.07 s	8.8 hr	106
Vocoder	arch1	10972	10980	170 (89 (0)/81)	0.27 s	17.0 hr	204
	arch2	11386	11415	223 (176 (0)/147)	0.34 s	22.3 hr	268
	arch3	11263	12276	559 (286 (0)/273)	0.43 s	55.9 hr	671
	arch4	13986	14033	369 (208 (0)/161)	0.45 s	36.9 hr	221

lated by $160 = (13.3 \text{ hr.})/(0.08 \text{ sec.} + 1 \text{ bus} \times 5 \text{ min.})$. With this assumption, the productivity gain is around several hundred times as a result of automatic refinement.

6. Conclusions

In this report, we presented a methodology and algorithms to automatically generate an intermediate communication link model from architecture model of a system. The network synthesis implements end-to-end communication semantics between system components which is mapped into point-to-point communication between communication stations of network architecture. The implementations of presentation layer for each data transaction are inlined into components during the network synthesis.

Network refinement tool, *scnr* has been developed and integrated into our SoC design environment. Experiments were performed to validate this concept. Simulations were done on input models and output models to ensure their semantic equivalence. Using an industrial-strength example, the feasibility and benefits of the approach have been demonstrated and several hundred times of productivity gain is obtained with network refinement tool.

References

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[Ger03] Andreas Gerstlauer. Communication abstractions for system-level design and synthesis. Technical Report CECS-TR-03-30, Center for Embedded Computer Systems, University of California, Irvine, October 2003.

[GG96] Michael Gasteier and Manfred Glesner. Bus-based communication synthesis on system-level. In *Proceedings of the International Symposium on System Synthesis*, pages 65–70, November 1996.

[GMG98] Michael Gasteier, Michael Münch, and Manfred Glesner. Generation of interconnect topologies for communication synthesis. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 36–42, March 1998.

[GYG03] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS modeling for system level design. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 130–135, March 2003.

[LRD00] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Efficient exploration of the SoC communication architecture design space. In *Proceedings of the International Conference on Computer-Aided Design*, pages 424–430, November 2000.

[OB98] Ross B. Ortega and Gaetanno Borriello. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 437–444, November 1998.

[PDN99] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimization and Exploration*. Kluwer Academic Publishers, 1999.

[Pen04] Junyu Peng. *System-Level Automatic Model Refinement*. PhD thesis, University of California, Irvine, Information and Computer Science, April 2004.

[YGG03] Haobo Yu, Andreas Gerstlauer, and Daniel D. Gajski. RTOS scheduling in transaction level models. In *Proceedings of the International Symposium on System Synthesis*, pages 31–36, October 2003.

[YW95] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 288–294, November 1995.