Protocol Generation for Communication Channels †

Sanjiv Narayan
Viewlogic Systems Inc.
Marlboro, MA 01752

Daniel D. Gajski
Dept. of Computer Science
Univ. of California, Irvine, CA 92717

Abstract

System-level partitioning groups processes and variables in the system specification into modules representing chips and memories. Communication between the modules is represented by abstract communication channels, which are merged and implemented as a bus to minimize interconnect. Given a set of channels, bus generation synthesizes the bus structure, by trading off the the width of the bus and the performance of the processes communicating over it. For each channel, we describe a method to generate protocols that specify the mechanism of data transfer over the bus. Protocol generation presented in this paper results in a refined system specification that is simulatable. Both bus-generation and protocol-generation are demonstrated on detailed examples.

1 Introduction

A system can be viewed as a set of *processes* which communicate with each other over *channels*. A channel is an abstract communication medium over which two processes can transfer data. System partitioning [1] may group processes and variables in the system specification into modules. The set of tasks which are performed to implement communication between the modules in a system are collectively defined as *Interface Synthesis*. For example, process A in Figure 1 is mapped to two system modules after system partitioning. The variables MEM and $STATUS$ which were originally declared within process A are now mapped to processes $A1$ and $A2$ respectively in a different module. Process A processes reads and writes data to the variable MEM over channels *ch1* and *ch2* respectively. $STATUS$ is accessed over channel $ch3$. In addition, to minimize the interconnect cost between the system modules, system partitioning may group channels to be implemented as a single bus. For example, in Figure 1, channels *ch1, ch2* and *ch3* have been grouped into a single bus, *B*.

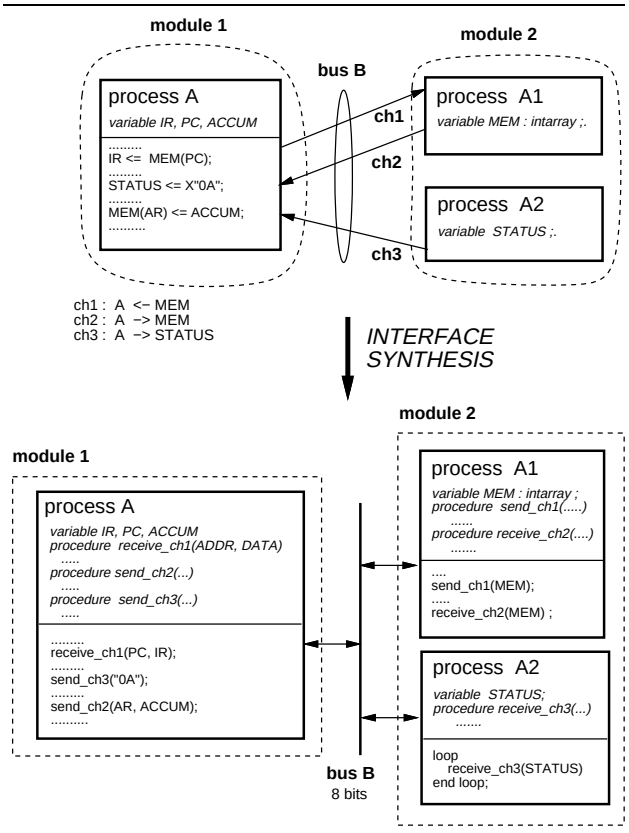

Figure 1: Interface synthesis in overall system design process

A channel is a virtual entity and free of any implementation details. After interface synthesis, a set of channels is implemented as a bus consisting of a set of wires and a protocol defining some behavior over the wires. Given a group of communication channels to be implemented as a bus, the goal of *interface synthesis* is to synthesize the bus structure and protocol with a view to minimizing the interconnect and maximizing the performance of the processes communicating over the bus.

Most previous research efforts that have examined interfaces can be classified into two broad cate-

†This work at U.C. Irvine was supported by the Semiconductor Research Corporation (grant #92-DJ-146).

gories. In the first category are research efforts which have incorporated interface timing constraints into the scheduling of operations and events within the process during high-level synthesis. These include ISYN [2], CONSPEC [3], and Interface Matching [4]. The second set of approaches address issues relating to interfacing standard components which have incompatible protocols. Synthesis of transducers was presented in [5, 6] where, given timing diagrams of two incompatible interfaces, a template matching strategy is used to assign hardware to behavior. In [7], the two behaviors being interfaced were specified as Verilog based FSMs and a cross product of the two FSMs was optimized to obtain the description of the transducer.

While channels merging was addressed in [4], it was assumed that all the channels transferred data of identical bitwidths. In our case, we wish to synthesize a bus to implement communication between different processes communicating over abstract channels created as a result of system partitioning. In addition, we seek to examine issues relating to design parameters external to the process (such as the number of pins or data transfer rates and their effects on performance of processes). None of the above approaches address such issues.

In Section 2, we formulate the interface synthesis problem. Bus and protocol generation are discussed in Sections 3 and 4. Results of our experiments with interface synthesis are presented in Section 5. Finally, we present our conclusions and plans for future work.

2 Problem Formulation

In implementing a group of channels, the goal of interface synthesis should be to synthesize a bus which has a 100% utilization, i.e., *the bus is never idle*. Consider two channels A and B which transfer data as shown in Figure 2. For simplicity, assume that the 4 second time interval shown in the figure is representative of the data transfer over the lifetimes of the processes which communicate over channels A and B. The channel **average rate**, $AveRate(C)$ is defined as the rate at which data is sent over channel C over the lifetime of the processes communicating over it. Channels A and B in Figure 2 have average rates of 4 and 12 bits/second respectively. If channels A and B are merged into a single bus AB, then we can see that the bus needs to send data at the rate of 16 bits/second to be able to satisfy the data transfer requirements of the two original channels A and B. The data items transferred over the channels have been labeled to make it easier to associate them with the data transferred over the shared bus. Consider the data item labeled *B2* transferred at the t=1 second in the original channel B, which is now transferred on bus AB at t=1.5 seconds. While individual data transfers may be delayed due to bus access conflicts, the bits transferred over the individual channels before channel merging are *still* sent over the shared bus *in the same amount of time*.

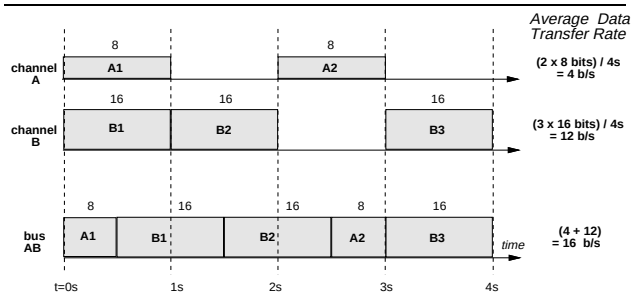

Figure 2: Merging channels A and B into bus AB.

In synthesizing the bus AB in Figure 2, we take advantage of the fact that the individual channels will not always be transferring data. We attempt to utilize the idle time slots of one channel for data transfers of other channels by synthesizing a bus over which data is *always* being transferred at a constant rate.

If the channels before being merged into a bus were transferring data at a certain average rate, they should be able to transfer the data over the bus at the *same average rate*. This can be achieved if the data transfer rate $BusRate(B)$, of bus B, is greater than the sum of the individual channel average rates. Thus,

$$BusRate(B) \geq \sum_{C \in B} AveRate(C) \qquad (1)$$

Implementing a group of channels consists of two tasks: Given a set of channels to be implemented as a single bus and a set of constraints, interface synthesis consists of two tasks: (1) **bus generation** which determines the *minimum cost* buswidth which will satisfy the data transfer rates of individual channels, and (2) **protocol generation** which generates the protocols for data transfer over the bus for each channel.

3 Bus Generation

The bus generation algorithm determines the width of the bus required to implement a group of channels. The bus generation algorithm was presented in [8].

Intuitively, the algorithm examines a range of possible buswidths. For each buswidth, the bus rate and the channel average rates are computed. If the bus rate is greater than the sum of the channel average rates (as explained in Equation 1), then we have a feasible bus implementation. From the set of feasible bus implementations, each corresponding to a different buswidth, we select the one which has the least cost. Briefly, the algorithm consists of five steps:

(1) **Determine buswidth range**: The smallest buswidth examined by the Bus Generation algorithm is 1 and the largest buswidth examined is equal to the largest size of message sent by any channel.

For each bitwidth, $CurrBW$, in the range determined above, repeat steps 2 and 3.

(2) **Compute the bus rate**: The bus rate depends on the delay of the protocol that will be used to implement the data transfer. For a handshake protocol, we assume that the delay is two clock cycles, thus,

$$BusRate(B) = \frac{CurrBW}{2 \times ClockPeriod} \quad (2)$$

(3) **Determine average rates for each channel** For all channels, determine the average rate for the current buswidth being examined. Estimation of channel average rates was presented in [8].

If the bus rate, $BusRate(B)$, is greater than the sum of the average rates of all the channels, then we have a **feasible implementation** for the bus. Go to step 4. If the bus rate, $BusRate(B)$, is less than the sum of the channel average rates, the bus rate will not be able to satisfy the performance requirements of the channels. Go to Step 2 and try with the next buswidth in the range determined in step 1.

(4) **Determine the cost function for** $CurrBW$ For a given set of channels which have been grouped together to be implemented as a single bus, the designer can specify constraints and relative weights for the buswidth, the minimum/maximum values of the channel average and peak rates. The cost of a bus implementation is calculated as the sum of the squares of violations of each of the constraints, weighted by the relative weights specified for them.

(5) **Select the buswidth** If there were one or more feasible solutions that were determined at the end of Step 3, select the buswidth corresponding to the one with the least cost determined in Step 4. If there were no feasible solutions at the end of step 3 for all the buswidths examined, then an implementation for the group of channels is not possible. Any implementation for such a group of channels would progressively delay the processes communicating over the bus. Such a situation can arise when several channels that have very high average rate requirements are grouped together to be implemented as a bus. One solution to this problem would be to split the group of channels further to be implemented by more than one bus.

4 Protocol Generation

Once an appropriate buswidth has been selected to implement the channel group, protocol generation defines the exact mechanism of data transfer over the bus. A bus consists of three sets of wires.

(1) **Data** lines are used to send data over the bus. The number of data lines (i.e., the buswidth) required can be determined by the bus-generation algorithm or they can be specified by the system designer. (2) **Control** lines are required for synchronization between the

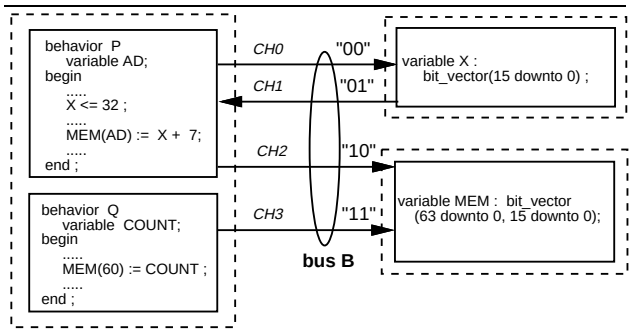

Figure 3: Behaviors accessing variables over channels.

behaviors that communicate over the bus. For example, a standard handshake protocol requires two control signals $START$ and $DONE$. (3) **Identification** or mode lines are required to uniquely identify the channel that is transferring data over the bus at any point of time. Since the bus control signals are shared by all channels, such identification (ID) lines are essential to enable behaviors to recognize when the control signals over the bus are meant for them. Each channel in the bus is assigned a unique ID which serves as the address of the channel.

We shall illustrate protocol generation through a simple example shown in Figure 3. Variables X and MEM are accessed by behaviors P and Q. The dashed lines indicate the assignment of the behaviors and variables to system components. Channels *CH0, CH1, CH2* and *CH3* are grouped into a single bus B whose width has been determined to be 8 bits. Protocol generation consists of five steps.

1. Protocol selection: Different communication protocols may be selected for a bus implementation, such as a full-handshake, half-handshake, fixed-delay and even hardwired ports. Each of the protocols require a different number of control lines. For bus B in Figure 3, a full handshake protocol is selected. Two control signals, $START$ and $DONE$, are used to implement the handshaking.

2. ID assignment: If there are N channels implemented on the same bus, $log_2(N)$ lines will be required to encode the channel ID. Unique IDs are assigned to each channel. The four channels in Figure 3 require 2 ID lines. Channel $CH0$ is assigned the ID "00", $CH1$ is assigned the ID "01" and so on.

3. Bus structure and procedure definition: The structure determined for the bus (i.e. the data, control and ID lines) is defined in the specification. For each channel mapped to the bus, appropriate *send/receive* procedures are generated, encapsulating the sequence of assignments to the bus control, data and ID lines to execute the data transfer. Figure 4 shows the declaration of an 8 bit bus, with 2 control lines and 2 ID lines. The bus, B, is declared to be a global variable (a signal in the case of VHDL) so that all behaviors can access it. Behavior P writes to the 16-bit variable

```
type  HandShakeBus is record
        START, DONE : bit ;
        ID : bit_vector(1 downto 0) ;
        DATA  : bit_vector(7 downto 0) ;
end record ;

signal  B : HandShakeBus ;

procedure  ReceiveCH0( rxdata : out bit_vector)  is
begin
    for J in 1 to 2 loop
        wait  until  (B.START = '1') and (B.ID = "00") ;
        rxdata (8*J−1  downto 8*(J−1)) <= B.DATA ;
        B.DONE <= '1' ;
        wait until  (B.START = '0') ;
        B.DONE <= '0' ;
    end loop;
end ReceiveCH0;

procedure  SendCH0( txdata : in bit_vector)  is
    bus B.ID <= "00" ;
    for J in 1 to 2 loop
        B.data <=  txdata(8*J−1  downto  8*(J−1)) ;
        B.START <= '1' ;
        wait until  (B.DONE = '1') ;
        B.START <= '0' ;
        wait until  (B.DONE = '0') ;
    end loop;
end SendCH0;
```

Figure 4: Defining bus B and send/receive protocols for channel $CH0$.

X over channel $CH0$. Since the buswidth in only 8 bits, procedures $SendCH0$ and $ReceiveCH0$ in Figure 4(b) transfer the 16-bit message associated with channel $CH0$ over the bus in two transfers of 8-bits each.

4. Update variable-references: References to a variable that has been assigned to another system component by system partitioning must be updated in behaviors that were originally referencing it directly. Accesses to variables are replaced by the corresponding send and receive procedure calls corresponding to the channel over which the variable is accessed. For example, in Figure 3, behavior P writes the value "32" to variable X directly. Channel $CH0$ represents the write to variable X. The statement ``X <= 32'' is replaced by the send procedure call ``sendCH0(32)'' as shown in Figure 5. The statement ``MEM(60) := COUNT'' in behavior $Q$ is updated to ``sendCH3(60, COUNT)'', indicating that the value in $COUNT$ is to be written to address 60 of array MEM.

5. Generate variable processes: In order to obtain a simulatable system specification, a separate behavior is created for each group of variables accessed over a channel. Appropriate send and receive procedure calls are included in the behavior to respond to access requests to the variable over the bus. In Figure 3, the variables X and MEM were assigned to different system components as shown by the dashed lines. In Figure 5, behaviors $Xproc$ and $MEMproc$ have been created for these two variables.

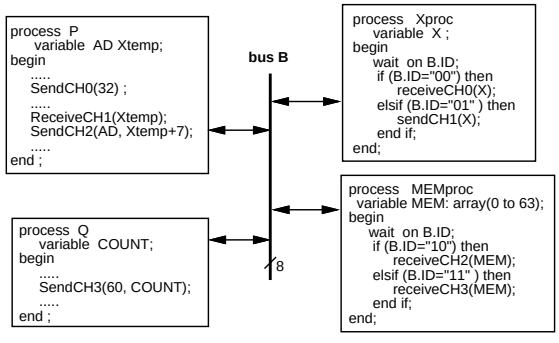

Figure 5: Refined specification after protocol generation.

5 Experiments and Results

Interface synthesis presented in this paper has been been implemented and integrated with the system-level partitioner presented in [1]. The partitioner groups the variables, behaviors and channels in a system specification into memories, modules, and buses respectively. We performed several experiments involving the application of the bus generation algorithm to synthesize module interfaces in an answering machine, an Ethernet network coprocessor and a fuzzy logic controller [9]. We will present the results for the fuzzy logic controller (FLC) in greater detail.

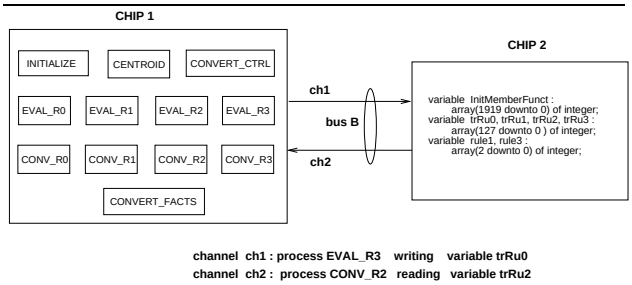

Figure 6: Processes and variables in the FLC partitioned into two chips

We shall illustrate the Bus Generation algorithm with the example of interface synthesis in the Fuzzy Logic Controller [9] of Figure 6. The Fuzzy Logic Controller consists of two inputs which sense the temperature and the humidity in a room. Depending on these two inputs, the FLC has 4 rules which are evaluated to compute the output signal which determines the operation of the air conditioning system. System partitioning mapped the memories (array variables in the description) which store the membership functions and fuzzy logic rules in the FLC to a separate chip. Processes $EVAL\_R3$ and $CONV\_R2$ of the fuzzy logic controller (FLC) access the array variables $trRu0$ and $trRu2$ respectively over communication channels $ch1$ and $ch2$ that have been merged to be implemented as a single bus.

Figure 7 shows how the performance of the two processes transferring data over bus B is affected by the various bus widths that can be used to implement the bus. For each bus width, the protocol required for each channel in the bus was generated. A performance estimator [10] was then used to obtain the execution times of the processes. Clearly, as the bus width increases, the execution time for the processes decreases. Since the two channels each transfer 16 bits of data

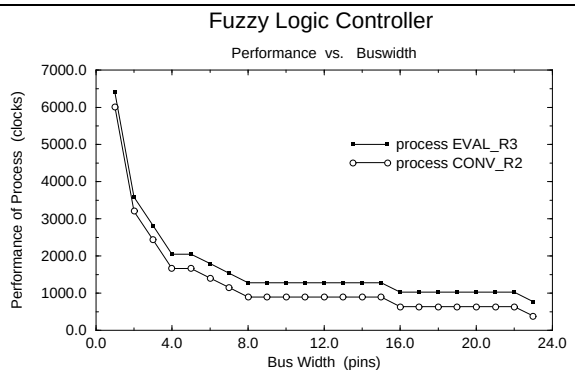

Figure 7: Effect of bus width on the execution time of processes *EVAL_R3* and *CONV_R2*

and 7 bits of address, bus widths greater than 23 pins do not yield any further improvements in the performance as the data transfer cannot be parallelized any further. If any performance constraints exist for these processes, the designer can select an appropriate buswidth for implementing the bus. For example, by examining Figure 7, if process *CONV_R2* has a maximum execution time constraint of 2000 clocks, then only buswidths greater than 4 bits will be considered by the designer during interface synthesis for implementing the bus.

Design Implementation	Bus Constraints (relative weight)	Total Bitwidth of the channels (pins)	Selected Bus Rate (bits/clock)	Selected Buswidth (pins)	Interconnect reduction (%)
A	Min PeakRate (ch2) = 10 bits/clock (10)	46	10	20	56 %
B	Min PeakRate (ch2) = 10 bits/clock (2) Min BusWidth (B) = 14 bits (1) Max BusWidth (B) = 16 bits (1)	46	9	18	61 %
C	Min PeakRate (ch2) = 10 bits/clock (1) Min BusWidth (B) = 16 bits (5) Max BusWidth (B) = 16 bits (5)	46	8	16	66 %

Figure 8: Bus constraints, selected bus width and corresponding bus rates of three implementations of bus comprising $ch1$ and $ch2$

To demonstrate how the designer can exercise control over interface synthesis by specifying appropriate constraints and weighing them accordingly, the Bus Generation algorithm was applied with three different sets of constraints. Figure 8 shows the bus constraints, selected bus widths, corresponding bus rates for the three bus designs A, B and C. In each case, specifying and weighing the constraints appropriately, the designer can implement the channel group with a different buswidth For example, in **design A** of Figure 8, the designer has specified a minimum peak rate for channel $ch2$ of 10 bits/clock. The minimum cost function corresponds to a bus width of 20 which is then used to implement the bus. The reduction in the number of data lines compared to the case when the two channels would have been implemented separately is 56%. In all the three examples, this reduction has been achieved *without sacrificing any performance of the processes.*

6 Conclusions and Future Work

Communication channels in system-level synthesis are often grouped together to reduce the interconnect cost at the module boundaries in a system. In this paper we have presented a method to generate protocols for implementing a group of communication channels as a single bus.

Protocol generation presented in this section has several advantages. First, the refined specification is simulatable and the design functionality after insertion of buses and communication protocols can be verified. Second, by encapsulating data transfer over the bus in terms of send and receive procedures, the description of the behavior remains relatively uncluttered as compared to the situation that would arise if we were to insert the assignments for the control and data lines at each communication point in the behavior. Finally, if at a later stage another communication protocol is selected for communication over the bus, only the bus declaration and send and receive procedures need be changed. The descriptions of the behaviors in the system, including the send and receive procedure calls, remain unchanged.

The work presented in this paper can be extended in several ways. We plan to study ways in which two or more channels may transfer data simultaneously over the same bus by utilizing different sets of data and control lines. This would be useful in cases when no feasible solution can be found in the range of buswidths examined. Incorporating protocols other than a full handshake needs to be studied. In addition, further work is needed to examine the effect of bus arbitration delays on the performance of processes.

7 Acknowledgements

We would like to thank Frank Vahid for his assistance in integration of the SpecSyn system partitioner with the bus and protocol generation tools. We would also like to acknowledge the the helpful suggestions of the reviewers.

References

[1] F. Vahid and D. Gajski, "Specification partitioning for system design," in *Proceedings of the DAC*, 1992.

[2] J. Nestor, *Specification and Synthesis of Digital System with Interfaces.* PhD thesis, Carnegie Mellon Unversity., April 1987.

[3] S. Hayati and A. Parker, "Automatic production of controller specifications from control and timing behavioral descriptions," in *Proceedings of the DAC*, 1989.

[4] D. Filo, D. Ku, C. Coelho, and G. DeMicheli, "Interface optimization for concurrent systems under timing constraints," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 268–281, September 1993.

[5] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proceedings of the ICCAD*, 1987.

[6] G. Borriello, *A New Interface Specification Methodology and its Applications to Transducer Synthesis.* PhD thesis, University of California, Berkeley, May 1988.

[7] J. Akella, *Input/Output Performance Modeling and Interface Synthesis in Concurrently Communicating Systems.* PhD thesis, Carnegie Mellon Unversity., November 1991.

[8] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the EDAC*, 1994.

[9] Matsushita Matsushita Electric Works R&D Lab, "Fuzzy logic controller." private communication, 1993.

[10] S. Narayan and D. Gajski, "Area and performance estimation from system-level specifications." UC Irvine, Dept. of ICS, Technical Report 92-16,1992.