Co-Synthesis of Custom On-Chip Bus and Memory for MPSoC Architectures

Sujan Pandey Christian Genz Rolf Drechsler

Department of Computer Science
University of Bremen, Germany
{pandey,genz,drechsle}@informatik.uni-bremen.de

***Abstract*— The advancement in process technology has made it possible to integrate multiple processing modules on a single chip. As a result of this, there is a sharp increase of communication traffic on the communication bus architecture. In this case, the traditional single bus based architecture may fail to meet the real-time constraints. The major concern of the scaled technology is an effect of coupling capacitance due to the trend of shrinking pitches, i.e., the distance between two wires. Its consequence is higher crosstalk noise, which degrades the signal integrity and modifies the power consumption of the wires. This motivates the synthesis of a custom on-chip bus architecture, which is efficient in terms of power and performance. Further, the memory of a complex multiprocessor system has a significant contribution to power and delay.**

In this paper, we present a co-synthesis of on-chip buses and memories, which finds an optimal bus architecture, memory sizes, and the number of memories. The bus synthesis problem is formulated as an optimization problem as proposed in [11], [9]. Then it is solved efficiently using an optimization tool. The memory synthesis problem is based on the graph partitioning algorithm, which partitions a data dependency task graph into a set of sub graphs with the minimum number of data dependencies called *cut*. The experiments carried out on the real-life multimedia applications validate the proposed technique for the co-synthesis of bus architecture and memory.

I. Introduction

Due to the advancement in process technology and the increasing demand of performance requirements for the next generation multimedia, broadband, and network applications, it is expected that by year 2009 more than 4 billion transistors will be integrated on a single chip according to the ITRS'05 roadmap [1]. As a result of these, more and more functionalities are being integrated onto a single chip which, in turn, results in a sharp increase of overall on-chip communication traffic among the integrated modules. In such complex systems, on-chip communication is expected to become a major performance bottleneck. Traditional approaches are mainly based on the synthesis of a single shared bus based architecture, which may not meet the real time constraints.

The early works on communication bus synthesis are mainly based on the simulation of an entire Hw/Sw system, which takes huge amount of time while exploring a large design space. The first approach for synthesizing a single global bus was proposed in [5], which finds the minimum bus width in order to minimize the chip size. In [14] an automatic bus generation for an MPSoC architecture was proposed. The approach considers for three different types of buses, which can be generalized to a shared bus, point-to-point, and FIFO based architecture. A bus architecture for a given bus width is generated considering real-time constraints. In [13] a method of communication synthesis based on the library elements and constraints graph was presented, where the library elements are a collection of communication links and communication nodes. The approach focuses mainly on synthesizing a communication bus topology for a point-to-point communication architecture. In [17] a bus model for communication in embedded systems with arbitrary topologies was proposed, where a point-to-point communication is a special case for the real-time application. Their algorithm selects the number of buses, the type of each bus, the message transferred on each bus, and schedules the communication bus. In [6] a synthesis flow which supports shared buses and point-to-point connection templates was presented. In [9], [10] an energy conscious on-chip bus synthesis technique was presented. All the above techniques are for synthesizing on-chip bus architectures, however, non of them synthesizes memories. Recently, in [12], an approach for co-synthesis of bus and memory was presented for MPSoC architectures. The technique synthesizes on-chip bus and memory, however, the synthesized bus architecture is not custom in terms of bus widths. It is rather based on the standard bus templetes provided by vendors.

In this paper, the bus synthesis algorithm is based on the approach proposed in [11], [9], which synthesizes a custom bus architecture. The synthesis problem is formulated as an optimization problem and finds the optimal bus widths and the number of buses. The memory synthesis is based on the graph partitioning algorithm, which clusters a set of tasks that have data dependencies. While partitioning a graph, the algorithm finds the minimum number of cuts among the clusters in order to minimize the communication via a bridge. This, in turn, results in the clusters of tasks, which seldom access the memory of another bus. The term cut is the number of edges connecting the clusters. It determines how many times an on-chip module accesses a memory using a bridge. Finally, the algorithm sums the data size of each cluster to find the memory size and maps each cluster onto a memory.

The reminder of this paper is organized as follows. Section II introduces a motivational example for co-synthesis of bus and memory. Section III gives a mathematical formulation and optimization techniques for on-chip bus synthesis problem. The memory synthesis algorithm based on graph partitioning is described in Section IV. In Section V, we present experimental results to validate our method of on-chip bus and memory synthesis and finally, in Section VI, we give the conclusion of this work.

II. Motivational example

In this section we give a motivation for co-synthesis of on-chip buses and memories and show that the synthesized buses and memories are optimal in terms of 1.) bus widths and the number of buses and 2.) memory sizes and the number of memories, respectively. We consider a partitioned and mapped Hw/Sw system. Based on the partitioned and mapped system, a communication task graph $G_C(C, \Pi)$ with nine communication tasks and their data dependencies is extracted as shown in Fig. 1(a). In the figure, tasks $\{c_4, c_5, c_7\}$, $\{c_8, c_9\}$, $\{c_1, c_2, c_3\}$, and $\{c_6\}$ are initiated by on-chip modules M1, M2, M3, and M4, respectively. After scheduling of tasks $c \in C$ as shown in Fig. 1(b), 16 and 24-bit buses are synthesized.

978-1-4244-1710-0/07/$25.00 © 2007 IEEE

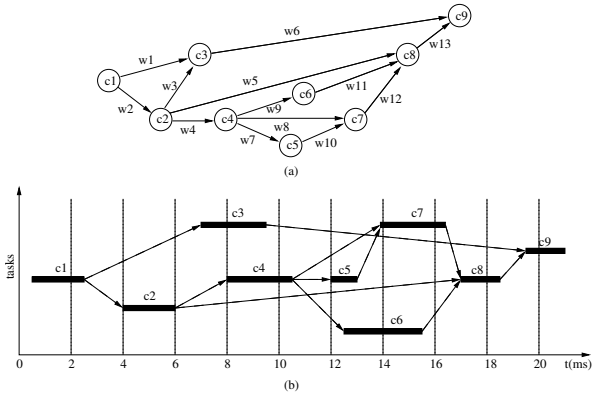

Fig. 1. Communication tasks and their schedule. (a) Example communication tasks. (b) The optimal schedule of tasks.

The tasks c_1, c_2, c_3, and c_6 are mapped to a 24-bit wide bus, while the tasks c_4, c_5, c_7, c_8, and c_9 are mapped to a 16-bit wide bus.

For the synthesis of memory sizes and the number of memories, we first extract an undirected data dependency tasks graph $G_D(C, \Pi)$ from the directed communication task graph $G_C(C, \Pi)$. In graph $G_D(C, \Pi)$, each vertex is a communication task c, while an edge between two vertices gives a data dependency between two communication tasks. From the data dependency tasks graph $G_D(C, \Pi)$, we find the maximum cliques with edges connecting each clique as shown in Fig. 2(a). The main aim is to cluster the data associated with tasks $c \in C$, which have data dependencies and map each cluster to a memory. From the given cliques of tasks $c \in C$ as shown in Fig. 2(a), we further cluster the cliques in order to find the minimum number of memories unless there is a memory access conflict (when two tasks access a memory at a same time). Fig. 2(b) and (c) show the synthesized memories and data associated with communication tasks. In the first figure, data of tasks $c \in C$, which are initiated by modules M1 and M2 are mapped to MEM1. While data of tasks initiated by modules M3 and M4 are mapped to MEM2. In the figure, there are three cuts, which mean that either on-chip modules M1 and M2 or M3 and M4 access memory MEM2 or MEM1 for three times using a bridge. Similarly, Fig. 2(c) depicts the synthesized memory sizes, number of memories, and the number of cuts. In the figure, the number of cuts is less than the synthesized memory of Fig. 2(b). This, in turn, results in less power and delay overhead due to communication via a bridge. Thus, the synthesis results of Fig. 2(c) give the optimal memory sizes and the number of memories with the minimum number of bridge accesses. The memory size is evaluated by summing all tasks $c \in C$ in a cluster. The synthesized on-chip buses and memories with their interconnection are shown in Fig. 2(d).

III. Bus synthesis

The on-chip bus synthesis problem is formulated as an optimization problem, which performs scheduling, allocation, and binding of tasks $c \in C$ and finds the optimal bus widths and the number of buses. During the scheduling, the slack of each task $c \in C$ is exploited to share the bus(es). The formulation of the optimization problem is given as follows [11], [9],
minimize:

$$\sum_{r \in R} Cost_r \cdot r_i \tag{1}$$

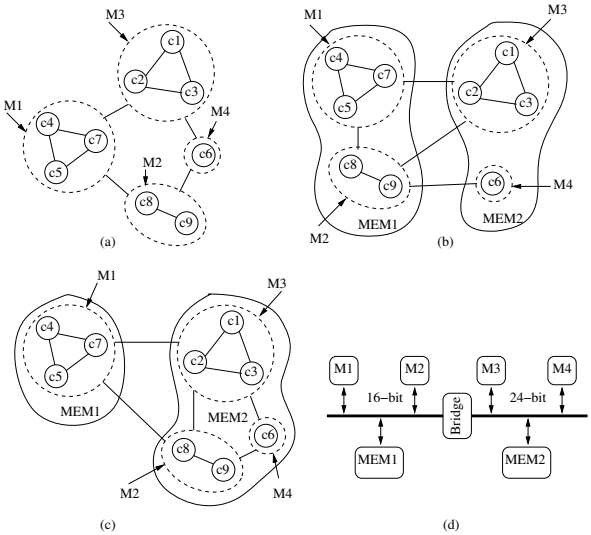

Fig. 2. Co-synthesis of on-chip buses and memories. (a) Clique of data dependency tasks and their dependencies. (b) Synthesized memories with number of cuts = 3. (c) Synthesized memories with number of cuts = 2. (d) Synthesized bus architecture and memories with interconnection of on-chip modules and bridge.

subject to,

$$s_\tau + w_\tau \leq dl_\tau \ \forall \tau \in T \tag{2}$$

$$s_\tau \geq s_{c'} + CLTI_{c',r} \cdot X_{c,t,r} \ \forall (c, c') \in \Pi \tag{3}$$

$$t \cdot X_{c,t,r} \geq (s_{\tau'} + w_{\tau'} \cdot X_{c,t,r} \ \forall \ (c', c) \in \Pi \tag{4}$$

$$(dl_c - s_c - CLTI_{c,r}) \cdot X_{c,t,r} \geq 0 \tag{5}$$

$$w_\tau = \sum_{\tau \in T} NC_\tau \cdot T_d \tag{6}$$

$$CLTI_{c,r} = \left\lceil \frac{NB_c}{b_r} \right\rceil \cdot T_d \tag{7}$$

$$T_d = \frac{\kappa_3 \cdot V_{dd}}{[\kappa_1 \cdot V_{dd} + \kappa_2 \cdot V_{bs} - V_{th}]^\alpha} \tag{8}$$

$$b_{r_{min}} \leq b_r \leq b_{r_{max}} \tag{9}$$

The main objective is to minimize the communication bus cost (bus widths and the number of buses) as shown in Eq. (1), where $r_i \in R$ is a library of on-chip communication buses with different bus widths. The $Cost_r$ of each bus r_i is expressed in terms of bus width. In Eq. (2), start time s_τ and execution time w_τ of each task τ should be less than or equal to its deadline dl_τ. Further, a task τ can start its execution only after its predecessor (communication task c) completes transferring data as shown in Eq. (3). A binary decision variable $X_{c,t,r} \in \{0, 1\}$, indicates scheduling of a communication task c at time $t \in \{0, \cdots, \lambda\}$, with bus width r. Eq. (4) gives a dependency between successor (communication task c) and predecessor (data processing task τ'). Since the delay $CLTI_{c,r}$ (communication lifetime interval) of a task c is a function of data size and bus width r (see Eq. (7)), Eq. (5) gives a delay constraint such that the overall delay of each task c must be less than or equal to deadline dl_c. The gate delay is calculated according to Eq. (8) [16], where V_{dd}, V_{bs}, and V_{th} are supply, body bias, and threshold voltages, respectively. Terms κ_1, κ_2,

κ_3, and α are the technology dependent parameters. While scheduling communication tasks for different bus widths r, Eq. (9) gives the constraint for bus widths. In the above formulation, the objective function is linear in optimization variable r_i and the delay constraint Eq. (7) is non-linear in the optimization variable r_i, thus, the above bus synthesis problem belongs to the convex quadratic optimization problem, which finds a global optimal solution in a polynomial time complexity [8].

IV. Memory Synthesis

The memory synthesis algorithm presented in Algorithm 1, takes an undirected data dependency task graph $G_D(C, \Pi)$ as input, which is extracted from a communication task graph $G_C(C, \Pi)$. The algorithm finds the optimal memory sizes and the number of memories with the minimum number of cuts. The term cut is the number of edges that connect the clusters. It determines how many times an on-chip module accesses a memory using a bridge. Thus, the aim is to map the data of communication tasks to an individual memory in order to minimize the number of communications between a module and a memory via a bridge. This, in turn, reduces power and delay overhead due to the communication. The clustering of tasks is based on the graph partitioning problem called clique partitioning [15], which finds a set of cliques from the given data dependency task graph $G_C(C, \Pi)$.

At line 1 and 2, the algorithm reads the data dependency task graph and the synthesized number of buses. From line 4 to 13, a super graph $G^{'}(S, E^{'})$ is derived from the graph $G_D(C, \Pi)$. Each node $s_i \in S$ is a super node [15] that can contain a set of one or more vertices $c_i \in C$. Edge $E^{'}$ is identical to E except that the edges in $E^{'}$ link to super nodes in S. At line 4 and 5, the algorithm initializes sets S and $E^{'}$ to empty sets. From 6 to 9, each vertex $c_i \in C$ of $G_D(C, \Pi)$ is moved to a separate super node $s_i \in S$ of $G^{'}$. In the graph $G^{'}(S, E^{'})$ each vertex $c_i \in C$ represents a communication task. An edge $e_{i,j} \in E$ between two vertices c_i and c_j represents a data dependency. From line 15 to 38 the algorithm finds the cliques of the data dependency graph. The algorithm stays in the while loop defined at line 15 until the set $E^{'}$ is not empty. In this loop the algorithm finds the super node of the graph, where each super node consists of all the nodes in connected nodes s_{Num1} and s_{Num2} with the maximum number of common nodes. By definition a super node $s_i \in S$ is a common node of the two super nodes s_j, $s_k \in S$ if there exist edges $e_{i,j}$, $e_{i,k} \in E^{'}$. At line 21, the function CommonNode($G^{'}, s_i, s_j$) returns the set of super nodes that are common nodes of s_i and s_j in $G^{'}$. At line 22, if the returned number of common nodes is greater than the variable *MostCommons* then the content of the variable is updated at line 24. When the *MostCommons* nodes are found, from line 30 to 32, two super nodes are merged into a single super node, $s_{Num1Num2}$, which consists of all the vertices in s_{Num1} and s_{Num2}. From line 36 to 38, the algorithm adds edges among the super nodes. The variable *CommonSet* consists of all the common nodes of s_{Num1} and s_{Num2}. At line 39 of the algorithm, a new while loop starts to synthesize memory sizes and the number of memories from the available cliques of a data dependency task graph $G_D(C, \Pi)$. The loop is repeated until the number of cuts is minimum and there is no more memory access conflict. The function *ClusterClique* at line 41 clusters the cliques in order to minimize the number of memories. Each time the algorithm finds the maximum number of edges connecting two cliques and clusters them to make a new super node. The loop is repeated until there is no minimum number of cuts and there is no memory access conflict. After the completion of the while loop defined at line 39, the algorithm gives a set of super nodes, which are mapped to an individual memory. At line 45, the data size of all the communication tasks in a super node s_i are summed to find the memory size for each memory.

```
MemorySynthesis()
 1  G_D(C, Π) ← GetDataDepTasks();
 2  n ← GetNumOfBuses();
 3  /*Create a super graph G'(S, E') */
 4  S ← ∅;
 5  E' ← ∅;
 6  for c_i ∈ C
 7  do
 8     s_i ← {c_i};
 9     S ← S ∪ {s_i};
10     endfor
11  for e_{i,j} ∈ E
12  do
13     E' ← E' ∪ {e_{i,j}};
14     endfor
15  while E' ≠ ∅
16  do
17     /*Find S_{Num1}, S_{Num2} having most common node*/
18     MostCommons ← −1;
19     for e'_{i,j} ∈ E'
20     do
21        c_{i,j} ← |CommonNode(G', s_i, s_j)|;
22        if c_{i,j} > MostCommons
23          then
24               MostCommons ← c_{i,j};
25               Num1 = i; Num2 = j;
26               endif
27        endfor
28     CommonSet ← CommonNode(G', S_{num1}, S_{num2});
29     /*Merge S_{Num1} and S_{Num2} into S_{Num1Num2}*/
30     S_{Num1Num2} ← S_{Num1} ∪ S_{Num2};
31     S ← S − S_{Num1} − S_{Num2};
32     S ← S ∪ {S_{Num1Num2}};
33     /*Add edge from S_{Num1Num2} to super nodes*/
34     for s_i ∈ CommonSet
35     do
36        E' ← E' ∪ {e'_{i,Num1Num2}};
37        endfor
38     endwhile
39  while cuts ≠ minimum ∧ MemAccessConflict ≠ true
40  do
41     cluster ← clusterClique(S, E');
42     endwhile
43  for c_k ∈ cluster
44  do
45     memSize_i ← Sum(DataSize(c_k));
46     endfor
47  return
```

Algorithm 1: Memory synthesis algorithm.

V. Experimental Validation

We validate the effectiveness of the proposed technique using real-life multimedia applications, i.e., an audio decoder [2] and a speech recognition system [3]. The audio decoder includes four main decoding steps, which are inverse quantization, channel decoupling, reconstruct curve, and IMDCT. After manually partitioning and mapping of the decoder [11], the IMDCT was mapped to a single hardware and the rest of the functionalities were mapped to a processor. Furthermore, the raw audio data was mapped to a compact flash (CF) memory with a CF-interface and the extracted audio data was mapped to an audio buffer for streaming. Similarly, the second speech recognition system consists of three main components: front

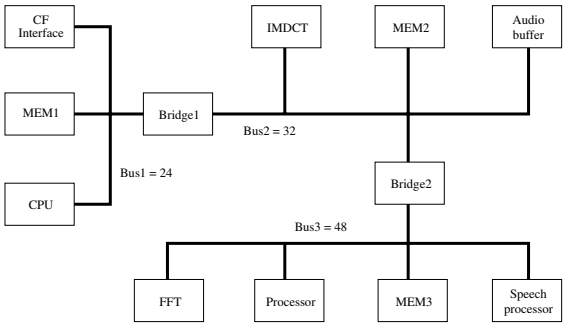

Fig. 3. Synthesized bus architecture with three buses and memory blocks.

end, decoder, and linguist. The front end includes series of data processing tasks such as pre-emphasis, hamming window, FFT (fast Fourier transformation), mel frequency filter, IFFT, cepstral mean normalization, and feature extraction to generate the features from the speech. The training takes as input a large number of speech along with their transcriptions into phonemes to provide the speech models for the phonemes. The recognition is based on the HMM (hidden Markov model) to decode the speech. We used the American English lexicon consisting of 32 phonemes and a database of 17 different words (spelling out the names of the months, numbers, and digits) [7], [11]. The length and the number of phonemes in a speech varies from application to application. After partitioning of the speech recognition system, the front end was mapped to a dedicated hardware including FFT and filters. The task training and recognition were mapped to a PowerPC processor. Based on the partitioned and mapped system, communication tasks and their data size were extracted by profiling the Hw/Sw system [7].

The on-chip communication buses are given as a library of buses with different bus widths, which ranges from 16 to 128-bit wide. The bus synthesis algorithm was implemented in C as a pre-processing model to interface with a convex solver of MOSEK [4]. Further, the memory synthesis algorithm was implemented in C as a graph partitioning problem, which partitions a set of communication tasks (associated with data) into a set of sub graphs with the minimum number of cuts among the sub graphs. For a given partition with the minimum number of cuts, each sub graph is mapped to a memory block. Fig. 3 shows the synthesized number of buses, memory blocks, and the interconnection of on-chip modules with buses. There are two bridges in order to communicate modules of one bus to the modules of another bus. The synthesized bus widths are 24, 32, and 48-bit wide. Table I shows the size of each memory block and the number of cuts between memories. In the column entitled *No. of cuts* of the table, MEM1 and MEM2 have 6 cuts, MEM2 and MEM3 have 10 cuts, while MEM1 and MEM3 have no cut. i.e., the smaller the number of cuts the better the performance in terms of delay and power overhead due to the communication through a bridge. Thus, the data associated with communication tasks mapped to MEM1 has no data dependency with the data associated with the communication tasks, which are mapped to MEM3. However, the tasks that are mapped to MEM1 and MEM2, and MEM2 and MEM3 have data dependencies.

Mem. Block	Mem. Size	No. of Cuts
MEM1	1.8 KB	(MEM1 and MEM2)=6
MEM2	2.4 KB	(MEM2 and MEM3)=10
MEM3	3.7 KB	(MEM1 and MEM3)=0

TABLE I
SYNTHESIZED MEMORIES, THEIR SIZE, AND NUMBER OF CUTS.

VI. CONCLUSION

Traditional on-chip bus synthesis approaches are mainly based on the simulation of an entire Hw/Sw system. After simulation, the communication requirement is mapped to a bus architecture provided by a vendor. However, the resulting synthesized bus architecture may not be efficient due to the under utilization of bus resources. In this paper, we proposed a custom on-chip bus architecture and memory co-synthesis techniques for a given application. The bus synthesis is formulated as a scheduling, allocation, and binding problems, and finds the optimal bus widths and the number of buses. The memory synthesis is formulated as a graph partitioning problem, which takes a data dependency task graph and synthesizes memory sizes and the number of memories. The experiments conducted on the real-life multimedia applications validate the effectiveness of the proposed technique.

REFERENCES

[1] http://www.itrs.net.
[2] http://www.xiph.org.
[3] http://www.speech.cs.cmu.edu/sphinx/.
[4] http://www.mosek.com/documentation.html#manuals.
[5] M. Gasteier and M. Glesner. Bus-based communication synthesis on system level. *In ACM Tran. of design automation electronic systems*, pages (1–11), 1999.
[6] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya. Automatic generation of application specific architectures for heterogeneous mpsoc. In *proc. of Design Automation Conference (DAC)*, pages (518–523), 2001.
[7] Z. Ming. Architecture exploration for speech-feature-extraction acceleration. *Bachelor thesis, Institute of Microelectronics Systems, Darmstadt University of Technology, Darmstadt, Germany*, September 2005.
[8] Y. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming*. Studies in Applied Mathematics, 1994.
[9] S. Pandey and M. Glesner. Statistical on-chip communication bus synthesis and voltage scaling under timing yield constraint. In *proc. of Design Automation Conference (DAC)*, pages (663–668), 2006.
[10] S. Pandey and M. Glesner. Simultaneous on-chip bus synthesis and voltage scaling under random on-chip data traffic. *In IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2007.(accepted for publication).
[11] S. Pandey, M. Glesner, and M. Mühlhäuser. Performance aware on-chip communication synthesis and optimization for shared multi-bus based architecture. In *ACM proc. of SBCCI*, pages (230–235), 2005.
[12] S. Pasricha and N. Dutt. COSMECA: Application specific co-synthesis of memory and communication architectures for mpsoc. In *proc. of Design Automation Test in Europe (DATE)*, pages 700–705, 2006.
[13] A. Pinto, L. P. Carloni, and A. V. Singiovanni. Constraint driven communication synthesis. In *proc. of Design Automation Conference (DAC)*, pages (783–788), June 2002.
[14] K. K. Rye and V. MooneyIII. Automated bus generation for multiprocessor soc design. *IEEE Tran. of Computer-Aided Design (CAD) of Integrated Circuits and Systems*, Vol. 23(No. 11):(1531–1549), Nov. 2004.
[15] C. J. Tseng and D. P. Siewiorek. Automated synthesis of data paths on digital systems. *IEEE Tran. of Computer-Aided Design (CAD) of Integrated Circuits and Systems*, Vol. CAD-5(No. 3):(379–395), 1986.
[16] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison wesley, 1994.
[17] T. Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pages (288–294), 1995.