Bus-Based Communication Synthesis on System-Level*

Michael Gasteier Manfred Glesner
Darmstadt University of Technology
Institute of Microelectronic Systems
Karlstrasse 15, 64283 Darmstadt, Germany

Abstract

We present 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, we first perform a clustering of data transfers. This results in groups of transfers suited to share a common bus. For each of these clusters we execute a bus generation algorithm which schedules bus accesses in order to minimize the total communication costs. Other than previous approaches, we infer RAM, if necessary, and consider data-dependencies as well as periodic execution of processes, like in VHDL. An example demonstrates the efficiency of the developed algorithm.

1. Introduction

Modern microelectronic systems consist of an increasing number of highly complex modules which are either realized on a single chip or distributed among multiple components. In the latter case, these components can be realized in different technologies, for example hardware components (ASICs, RAM etc.) and programmable processors (microcontrollers, DSPs or ASIPs). Such mixed hardware-software architectures can most often be found in *embedded systems* [11]. Designing those systems requires a new design methodology which is referred to as *hardware-software co-design*. One major research topic in co-design is the automatic generation of a communication topology adjusted to the requirements of a system to be realized, the so-called *communication-synthesis*. The goal of communication-synthesis is to find a minimum cost communication topology which guarantees the given system performance constraints to be fulfilled. Within the context of this paper, costs are defined as a weighted sum of the required bus widths and costs for intermediate storage.

The work presented here is part of our hardware-software co-design project DICE (**D**armstadt **I**nteractive **C**o-design **E**nvironment) [6]. At system-level, we use VHDL processes and C programs to describe the behavior of the system. These processes communicate with each other via abstract `send` and `receive` functions which allow asynchronous and synchronous data exchange. In order to validate the behavior of the mixed hardware-software description, the abstract communication functions enable co-simulation by coupling the execution of C code to simulation of VHDL using standard VHDL simulators [5].

During communication synthesis, we have to replace the abstract communication by a physical structure which allows data exchange at the required transfer rate. A very cheap implementation can be achieved by using buses without any arbitration scheme which, however, requires a deterministic control flow to be able to statically schedule all bus accesses in advance in order to avoid access conflicts. Our approach is based on this assumption, but can be extended to handle non-deterministic control flow caused by data dependent execution or synchronization. Such points of non-determinism are usually introduced by control signals which start and stop certain processes. Even if the timing of these control signals cannot be predicted in advance, the code portions between such points of non-determinism are scheduled statically, allowing to use buses without arbitration for all transfers executed within such sections. Additional hardware, for example separate handshake lines, has to be used for implementation of synchronization required by such control signals.

The algorithm presented in this paper determines a minimum cost communication topology for a set of statically scheduled processes. The abstract communication used at system-level is replaced by one or more buses connected to additional RAM, if required. The resulting bus accesses are scheduled within the processes under consideration of data dependencies in order to minimize the necessary bus widths.

2. Related Work

The research activities published in the area of communication synthesis can be grouped into three categories.

Most work done so far relates to synthesis of dedicated hardware for different communication aspects. Automatic

*This work was supported by the DFG under grant Gl144/11-1

generation of software and hardware according to a given communication protocol [10] belongs into this category as well as synthesis of devices for interfacing incompatible protocols [3] or sizing of communication queues [2]. The second category comprises approaches for scheduling of transfer operations like bus accesses during high level synthesis in order to satisfy given timing constraints for I/O operations, like the CONSPEC system [7].

Common to all publications mentioned so far is the limitation to single channels. None of the approaches above is intended to be used on system-level, thus allowing a global optimization of a complete communication topology. Up to now there are only a few publications which try to optimize communication between a set of processes incorporating techniques like channel merging. NARAYAN and GAJSKI presented a bus generation algorithm which determines a minimum bus width for a given set of communication channels [8]. YEN and WOLF map a multiple process description onto a number of processing elements (PEs) [12]. Communication between these PEs is executed via buses, connecting a PE to a bus increases the total communication costs. Priorities are assigned to data transfers in order to solve access conflicts while minimizing the overall communication delay times.

The interface optimization system presented by FILO ET AL. [4] schedules transfer operations simultaneously during high-level synthesis, trying to realize communication by point-to-point connections not considering RAM access. If this is not possible due to missing overlap of the send and receive intervals, they block a process until communication can be executed.

Other than the approaches by NARAYAN ET AL. and YEN ET AL. which assume buses with arbitration to be used, we try to minimize communication cost by inferring buses which do not need an arbitration scheme. This requires scheduling of bus accesses and assignment of specific bus lines for each transfer in order to avoid parallel write accesses under consideration of data dependencies.

3. Problem Description

As already mentioned in the introduction, the approach presented here allows generation of very cost efficient communication topologies for mixed hardware software systems with deterministic control flow. The processes have to be executed at the same clock speed and the data transfer characteristics have to be known in advance. For systems which do not fulfill the deterministic control flow criteria, flow analysis techniques can be used in order to identify subparts of the system and transfers in between for which this criteria holds. If identification of such regions is not possible or system components running at different clock rates are involved, other approaches, for example the ones proposed by YEN and NARAYAN, have to be incorporated.

Throughout this paper we use a small example shown in listing 1 to illustrate our approach. A more advanced example will be presented in section 5. For clarification of our approach, the small example does not contain any software processes, since timing in VHDL descriptions is explicitly determined by wait statements, thus showing clearly the scheduling of the operations. This is not the case for software programs which have to be compiled in order to analyze the timing of external input/output. Of course, our approach can be applied to software systems or mixed hardware-software systems as well, as long as the exact assignment of operations to clock cycles can be determined in advance.

Listing 1 A small demonstrator

```
SendP: process                      RecP: process
begin                               begin
  BS := 0;                            AR := 0;
  wait until Clk ='1';                recA("SendP", "T2", BR);
  AS := 3 + CS;                       wait until Clk ='1';
  sendA("RecP", "T1", AS);            wait until Clk ='1';
  wait until Clk ='1';                recA("SendP", "T1", AR);
  BS := AS + CS;                      CR := AR - BR;
  wait until Clk ='1';                wait until Clk ='1';
  sendA("RecP", "T2", BS);            wait until Clk ='1';
  AS := 0;                          end process RecP;
  wait until Clk ='1';
end process SendP;
```

As shown in listing 1, the example we use consists of two VHDL processes, `SendP` and `RecP`. Process `SendP` generates two data values, stored in variables `AS` and `BS`, which are transmitted via asynchronous send/receive functions to process `RecP`. `RecP` stores these values in `AR` and `BR`. Both processes are iterated with a period of four clock cycles. An additional identifier (`T1` and `T2` in the example) is added to the parameter lists of the transfer functions. So the send/receive paradigm used is not longer restricted to modeling FIFO-like communication channels only, but also connections requiring intermediate data storage.

Please note that the VHDL processes in the given example do not communicate via any global signals. Communication is completely done by using the abstract communication functions which would also allow to address a software process, e.g. written in C, in an identical manner [5].

3.1. Acquisition of Transfer Information

Starting with a single VHDL entity containing multiple processes and a set of (compiled) C programs, as required by our co-design environment, we apply data flow analysis [1] techniques and co-simulation in order to collect information about the data exchanged between each pair of processes. This information serves as input for the following communication-synthesis step. In particular we need for each transfer a complete identification (name of sending and receiving processes, identifier), the access mode (send/receive), the number of bits transferred and the exact timing(clock cycle during which transfer is executed).

During co-simulation we create a log file which contains these characteristics. However, as mentioned above, when using buses without arbitration scheme one has to ensure that no parallel write operations accessing overlapping *bus slices* occur during the life-time of all processes. The term bus slice is introduced here because we allow parallel write accesses as long as they do not address identical bus lines. It refers to a set of bus lines usually located next to each other. Since the lifetime of a process in VHDL is usually unbounded, this condition cannot be guaranteed by analyzing the log file generated during co-simulation, because the simulation is limited to a certain time range. We will therefore use *control/data flow analysis* techniques in order to derive the necessary transfer information. Up to now we rely on the log files generated during co-simulation. The designer has to ensure that the time range for which co-simulation is executed, is sufficiently large.

After retrieving the transfer information by co-simulation and/or control/data flow analysis, we perform a preprocessing step in order to convert the information into a more suitable format. During this preprocessing the following operations are executed:

Step 1: Analysis of mobility of each send and receive operation which is limited through data dependencies (this requires again a control/data flow analysis)

Step 2: Restriction of mobility ranges for each send and receive operation.

Step 3: Conversion into a new format containing information about mobility and mode of each transfer.

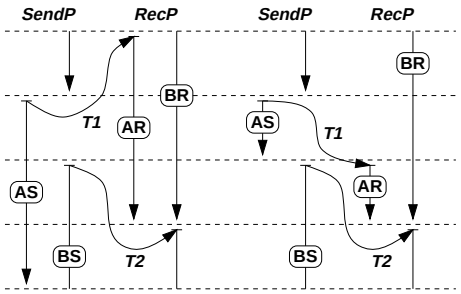

Figure 1. Small example illustrating approach

Figure 1(a) shows the mobility ranges identified by control/data flow analysis for the values to be transmitted. For example, value BS in process SendP becomes available in clock cycle three, and can therefore not be sent in an earlier clock cycle. Since we use variables for storage of data values, it would be possible to move the send function right behind the statement BS := AS + CS. When executing the next iteration of process SendP, BS will be overridden again in clock cycle one of iteration two which corresponds to clock cycle five after unrolling the iteration. However, during this clock cycle execution of the send operation would still be possible by moving the send function right in front of the statement BS := 0. In total we gain a mobility range from clock cycle three to clock cycle five for BS, as indicated in fig. 1(a). The mobility ranges of the other operations can be determined analogously.

In the second step, the resulting mobility ranges are restricted according to the following rules:

Rule 1: The end of the mobility range of a send operation can be limited to the clock cycle prior to the end of the mobility range of the corresponding receive function.

Rule 2: The beginning of the mobility range of a receive function can be limited to the clock cycle right behind the beginning of the mobility range of the corresponding send function.

Applying these rules leads to restricted mobility ranges as depicted in fig. 1(b). The mobility ranges for transmitting the value identified by T1 (AS resp. AR) can be limited from three clock cycles to one single clock cycle for both the send and the receive operation. For the second transfer, no restriction can be achieved.

Finally we determine a transfer mode which can be either one of the following:

IMM: immediate, the receive operation is executed one clock cycle behind the send operation. Considering the VHDL signal semantics which are also valid in our co-simulation, this means that writing and reading to/from the bus is executed simultaneously, not requiring intermediate storage.

SEND: send operation, where the receive operation has not necessarily to be executed simultaneously. If the mobility ranges of send and receive are completely disjoint, this will result in inference of memory and a memory write access. Otherwise both solutions (memory or simultaneous write/read) are considered.

REC: like SEND, but corresponding receive.

Thus, for the transfer of the value identified by T2 we get a pair of entries, one of mode SEND and one of mode REC, while the description of the transfer of T1 can be collapsed into one single entry of mode IMM. Table 1 shows the resulting transfer information, assuming that T1 has a size of eight bits while T2 needs 16 bits. An iteration number of zero means that a process iterates forever.

Mode	SenderID	RecID	Ident	Width	First	Last	Cycle	ItNbr
IMM	SendP	RecP	T1	8	2	2	4	0
SEND	SendP		T2	16	3	5	4	0
REC		RecP			4	7	4	0

Table 1. Transfer information for example

The format currently used limits our approach to periodically executed transfers, where a suspension of the transmission can be inserted after a number of iterations. This limitation can be relaxed by allowing a more complex description format. The communication synthesis algorithm in principle can handle arbitrary complex descriptions, adjustments would be necessary in small subparts only.

3.2. Basic Communication Structure

The basic communication structure is shown in fig. 2. Depending on the results of the communication synthesis, one or more of these bus structures are used for connecting the processes. In order to achieve a minimum bus width, multiple accesses can be executed simultaneously, as long as they address different bus lines.

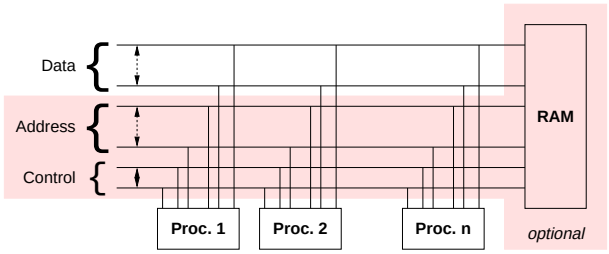

Figure 2. Basic communication structure

RAM components are only inferred if intermediate data storage is required due to disjoint mobility intervals of send and receive functions or if a reduction of the total communication costs can be achieved through reduction of bus width. Different types of RAM can be offered by the designer in a library from which the cheapest solution will be selected considering access protocols and other characteristics.

3.3. Additional Requirements

Scheduling of bus accesses is executed by replacing the abstract communication functions with bus line accesses, where these accesses have to be placed within the restricted mobility range of each `send`/`receive` function. During scheduling, four additional requirements have to be taken into account, increasing the complexity of the scheduling problem compared to the scheduling problem known from literature: (a) Due to the periodic execution of VHDL processes with possibly different cycle times, the schedule has to be executed for not only one iteration, but for a number of iterations large enough to guarantee that no conflicting bus accesses will occur. For each iteration, the access has to be scheduled at the same clock cycle relative to the beginning of the process. (b) Each transfer has to access the same bus slice at each iteration. (c) Data dependencies between send and receive operations have to be obeyed. (d) Accessing RAM usually requires multiple bus cycles, depending on the type of RAM.

The problem to be solved here belongs to the class of *time-constrained scheduling* problems.

4. Synthesis Algorithm

The algorithm used to handle the communication synthesis problem as described above executes two basic tasks:

Task 1: *Merging of transfers to clusters* which are to be realized on a common bus. This yields a system of buses to which we will refer in the following as *communication topology*.

Task 2: *Generation of a minimum cost bus structure* according to the model described in section 3.2 for each cluster.

Since these two tasks are not independent of each other, we perform an additional optimization step after bus generation. In the following we explain the algorithm, which is shown in listing 2, in more detail.

Step 1: Clustering of transfers

Since we assume periodic execution of our processes, like in VHDL, our clustering is based on the cycle length of the processes. The goal is to merge transfers which do not interfere with each other too much. This can best be done by joining processes with identical cycle length or where the larger cycle length is a multiple of the smaller cycle length. The worst case occurs when two processes P_1 and P_2 with cycle lengths $CL(P_1)$ and $CL(P_2)$ are joined, where $gcd(CL(P_1), CL(P_2)) = 1$. In this case, each bus access of P_1 will be executed in parallel to each access of P_2 at some point of execution.

Let P^A be the set of processes executing transfers merged in cluster A, and P^B the analog set of processes for cluster B. In order to reduce the number of generated clusters, in a second step we merge clusters A and B, if

$$\forall (P_i^A, P_j^B) \in P^A \times P^B : \text{gcd}(P_i^A, P_j^B) > 1 \qquad (1)$$

In listing 2, the clustering is executed by the function *generateClusters(TrList)*. The resulting clusters are collected in $CList$.

Step 2: Bus generation

In this step, a minimum cost bus implementation is generated for each cluster. We use a *branch-and-bound* approach to solve this task which in principle evaluates the total communication costs for each possible schedule. Neglecting any data dependencies between the transfer operations, the number of combinations equals the product of all mobility range lengths of the transfers contained in the current cluster. Although this number decreases, when data dependencies are introduced it is still a computationally highly intensive task. Computation time can be further reduced by cutting as soon as possible branches of the search tree which implicitly enumerates all possible schedules.

Listing 2 Communication synthesis algorithm

```
StatCommSynth() {
  initialize(TrList);
  CList = generateClusters(TrList);
  foreach (C_i ∈ CList) {
    calcScheduleLength(C_i); // LCM based
    scheduleTransfers(C_i);
  }
  optimizeResults(CList);
}

scheduleTransfers(C) {
  Cost_best := ∞;
  rearrange(C); // improve order
  Tr_first := getFirstTr(C);
  if (mode(Tr_first) = IMM)
    scheduleTrImm(Tr_first); // IMM transfers only
  else scheduleTrSend(Tr_first);
  if (Cost_best == ∞) return; // no feasible solution
  foreach (Tr_i ∈ C);
    mapToBus(Tr_i); // maps best solution found
}

scheduleTrSend(Tr_i) {
  if (⋂ mobRange(Tr_i) ≠ ∅) {
    // subrange of mobility for immediate transfer
    foreach (TimeSlot ∈ mobRangeMod(Tr_i)) {
      Slice_act = findSlice(Tr_i);
      if (Cost_curr < Cost_best) {
        mapToBus(Tr_i);
        skipReceivers(Tr_i);
      }
      if (∄Tr_i+1) checkCost(Cluster);
      else if (mode(Tr_i+1 == SEND)
        scheduleTrSend(Tr_i+1);
      else scheduleTrImm(Tr_i+1);
      unmapFromBus();
    }
  }
  foreach R_i ∈ RAMList {
    process like imm. transfer, but schedule multiple
    accesses (send and rec.) according to RAM
    interface and consider additional RAM costs
  }
}

scheduleTrImm(Tr_i) {
  foreach (TimeSlot ∈ mobRange(Tr_i); {
    Slice_act = findSlice(Tr_i);
    if (Cost_curr < Cost_best) {
      mapToBus(Tr_i);
    }
    if (∄Tr_i+1) checkCost(Cluster);
    else scheduleTrImm(Tr_i+1);
    unmapFromBus();
  }
}
```

In order to calculate the bus width required for a certain schedule, we map the bus accesses according to this schedule. A bus slice is assigned to each access under the constraints explained in section 3.3. The number of clock cycles, for which the mapping has to be executed, is calculated based on the least common multiple (LCM) method, as explained e.g. in [9] by calling the function *calcScheduleLength()*.

We use a recursive algorithm which creates a search tree in DFS (depth first search) order. A cost estimation function incorporating look-ahead techniques allows us to compare estimated costs of the current branch $Cost_{curr}$ at any depth-level to the minimum total costs $Cost_{best}$ achieved so far. If $Cost_{curr}$ exceeds $Cost_{best}$, this branch will probably not lead to cost improvement and will be cut. The efficiency of the branch-and-bound technique is improved, if "good" solutions are calculated first, because this allows early cutting of search branches. For transfers which can be either executed immediate or with intermediate storage of data, we first process the immediate solutions because these will usually yield lower costs than solutions inferring additional RAM.

The top level function used for bus generation is *scheduleTransfers()* which first optimizes the order of transfers by calling the function *rearrange()*. The functions *scheduleTrSend()* and *scheduleTrImm()* call each other recursively, thus traversing the search tree. When finding a solution yielding lower cost, the current schedule is stored in a global data structure. The function *findSlice()* searches a free bus slice for scheduling all accesses of a (periodically occurring) transfer. Mapping of accesses is executed by the function *mapToBus()* which also considers different RAM types.

Step 3: Final optimization

In order to optimize the results achieved by executing the second step, the function *optimizeResults()* tries to merge generated buses. Two buses are merged if the total bus-width of the merged bus would be lower than the sum of the bus widths of the two buses due to sharing bus lines.

The presented algorithm does not necessarily find an optimal solution since we do not perform a complete search but limit the search by applying two restrictions: (1) Imposing a search order which affects selection of RAM. The search order leads to optimal results with a high probability, but cannot guarantee them. (2) Performing quick look-ahead cost estimation. Overestimating costs could lead to cutting a branch of the search tree which contains the optimal solution.

However, experiments have shown that our approach finds the optimal solution in most cases. In all other cases the solution found was very close to the optimal solution.

5. Example Application

In the following we will present an example showing some results generated for a given cluster, assuming that the previous clustering was already executed. A process `DataGen` generates data which is transferred to two processes, `Filter` and `ChkFilter`. `Filter` processes the data and transfers its results to `Proc`. `ChkFilter` also accesses these results and compares them with the values generated by `DataGen`. The results of this compare are forwarded to process `Proc`. Furthermore, `Filter` sends some characteristics evaluated during filtering to `EvalChar`. `P1` and `P2` are two additional processes not involved into the filtering process but owning the same iteration rate. Details of the various data transfers can be found in table 2.

Assuming that RAM is available which requires two clock cycles for access, the output listed in table 3 is generated. A better overview is provided by a graphical display of the output, as shown in fig. 3.

Value `FData` has to be stored in RAM since the mobility range for receiving this value in process `ChkFilter` does

Mode	SenderPid	RecPid	Ident	Width	First	Last	Cycle	ItNbr	Pause
SEND	DataGen		Data	10	1	2	8	4	16
REC		Filter			2	4	8	4	16
REC		ChkFilter			2	6	8	4	16
IMM	Filter	EvalChar	Char	8	33	34	2	8	32
SEND	Filter		FData	16	4	6	8	4	16
REC		ChkFilter			7	8	8	4	16
REC		Proc			5	8	8	4	16
SEND	ChkFilter		Status	8	1	3	8	-	-
REC		Proc			2	4	8	-	-
IMM	P1	P2	ExtTr	8	3	4	16	-	-

Table 2. Transfer requirements of example application

PID	Transmission	Slice	first	RAM
Filter	send(FData)	B0-B15	4	SRAM2(0)
ChkFilter	rec(FData)	B0-B15	7	SRAM2(0)
Proc	rec(FData)	B0-B15	5	
DataGen	send(Data)	B0-B9	2	
Filter	rec(Data)	B0-B9	2	
ChkFilter	rec(Data)	B0-B9	2	
ChkFilter	send(Status)	B0-B7	3	
Proc	rec(Status)	B0-B7	3	
P1+P2	s&r(ExtTr)	B8-B15	3	
Filter+EvalChar	s&r(Char)	B0-B7	34	
SRAM2(0)(WR:2,RD:2)	B0-B15			

Table 3. Transfer requirements of example application

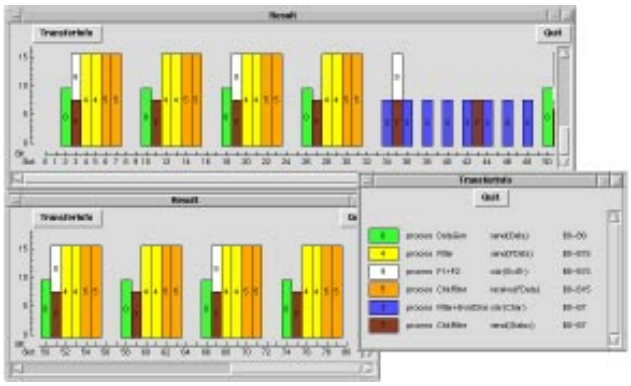

Figure 3. Graphical display of bus accesses

not overlap the mobility range of the corresponding send operation. Reading and writing this value thus requires two clock cycles. However, process `Proc` reads this value directly when it is written which is, according to the bus protocol, the second cycle of the RAM access. The transfer of `Data` was realized as immediate transfer since all mobility ranges involved overlap.

The resulting bus width required to realize this example is 16 bit. The calculated schedule length is 80 clock cycles. Solving this small problem requires 10 milliseconds on a SPARC-20. However, for more complex systems the runtime of the algorithm lies usually within the range up to a few seconds.

6. Conclusion and Further Work

Within this paper we have presented an algorithm for generation of low cost communication topologies for statically scheduled systems.

The next improvement we will introduce is an extension towards dynamic systems, where statically scheduled subparts will be realized using the presented approach. Furthermore, our description format will be extended in order to allow more complex control flow. Due to the more complex control flow we also have to improve the clustering technique. We will analyze the average usage of the abstract transfer channels and incorporate these results into the clustering process.

Some more optimization of our approach could be achieved for hardware processes by inserting latches into the VHDL processes, in order to extend the mobility range of the transfer operations. This will also lead to more efficient results.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] T. Amon and G. Borriello. Sizing Synchronization Queues: A Case Study in Higher Level Synthesis. In *Proc. of the 28th Design Automation Conference*, June 1991.

[3] G. Borriello and R. Katz. Synthesis and Optimization of Interface Transducer Logic. In *Proc. of the IEEE Transactions on CAD/ICAS*, November 1987.

[4] D. Filo, D. Ku, N. Coelho, and G. De Micheli. Interface Optimization for Concurrent Systems Under Timing Constraints. *IEEE Transactions on VLSI Systems*, 1(3):268–281, Sept. 1993.

[5] M. Gasteier and M. Glesner. Co-simulation of Mixed HW/SW Systems (orig: Cosimulation gemischter HW/SW-Systeme). In M. Glesner, editor, *Hardwarebeschreibungssprachen und Modellierungsparadigmen: 2. GI/ITG/GME-Workshop*, pages 60–69, Darmstadt, Februar 1996. Shaker Verlag.

[6] M. Glesner, M. Gasteier, and M. Münch. An Overview on Hardware/Software Co-Design. In *Proc. of MIXDES 96*, Lodz, May 1996.

[7] S. Hayati and A. Parker. Automatic Production of Controller Specifications from Control and Timing Behavioral Descriptions. In *Proc. of DAC '89*, 1989.

[8] S. Narayan and D. Gajski. Synthesis of System Level Bus Interfaces. In *Proc. of The European Design and Test Conference 94*, pages 395–399, Paris, France, February 1994.

[9] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. In *Proc. of Interntional Conference on Distributed Computing Systems*, 1990.

[10] A. Wenban, J. O'Leary, and G. Brown. Codesign of Communication Protocols. *IEEE Computer*, pages 46–52, December 1993.

[11] W. Wolf. Hardware-Software Co-Design of Embedded Systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[12] T.-Y. Yen and W. Wolf. Communication Synthesis for Distributed Embedded Systems. In *Proc. of ICCAD 95*, pages 288–294, San Jose, CA, Nov 1995.