In this article, we present an approach to automatic generation of communication topologies for statically scheduled systems or subsystems. Given a specification containing a set of processes that communicate via abstract send and receive functions, we show how a cost-efficient communication topology consisting of one or more buses without arbitration scheme can be set up for such applications.

Categories and Subject Descriptors: B.4.3 [Input/Output and Data Communications]: Interconnections (subsystem)

General Terms: Algorithms, Design

Additional Key Words and Phrases: Bus generation, bus without arbitration, communication synthesis, statically scheduled systems, transfer scheduling

1. INTRODUCTION

Modern microelectronic systems consist of an increasing number of highly complex modules that 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 and programmable processors. The quality of the interfaces and throughput of communication connections between these components is crucial to the performance of the complete system, since communication is often the main bottleneck in modern application domains like image processing and telecommunication. Possible implementation alternatives range from dedicated point-to-point connections to global buses. Due to cost and time-to-market pressure, it is usually not practical to manually evaluate all possible alternatives. Therefore communication synthesis tries to automatically determine a cost-efficient communication topology. Costs are, in this context, usually defined as a weighted sum of required bus widths, costs for
intermediate storage, and area for peripheral communication devices, like interfaces or arbitration logic. The work presented here is part of our hardware-software co-design project DICE (Darmstadt Interactive Co-design Environment) [Gasteier et al. 1996], which also provides tools for partitioning and mechanisms for VHDL/C co-simulation [Gasteier and Glesner 1996].

After summarizing some of the work related to communication synthesis published so far, we will briefly describe in Section 3 the environment in which the approach we are presenting here is integrated. The main focus of this article, the generation of very cost-efficient communication structures based on buses without arbitration scheme, will be presented in Section 4. This approach relies on the predictability of the execution time of each action on clock cycle level. We call these systems in which the control-flow does not depend on external data statically scheduled.

2. RELATED WORK

Up to now, there have been only few papers that try to optimize communication between a set of processes. Fundamental work was done by Filo et al. [1993]. Their interface optimization system schedules transfer operations simultaneously during high-level synthesis, trying to replace blocking transfers by nonblocking transfers in order to minimize hardware resources.

Narayan and Gajski [1994] presented a bus generation algorithm that determines a minimum bus width for a given set of communication channels to be realized on one bus. Yen and Wolf [1995] map a multiprocess description onto a number of processing elements (PEs). Communication between these PEs is executed via buses; connecting a PE to a bus increases the total communication costs. Chou et al. [1995] developed the CHINOOK system, which executes library-based communication synthesis. The CoWare system [Van Rompaey et al. 1996] realizes synchronized send and receive functions by using hierarchical channels.

Unlike the approaches by Narayan and Gajski and Yen and Wolf, which assume buses with arbitration to be used, we try to minimize communication cost by inferring buses that do not need an arbitration scheme. This requires scheduling of bus accesses and assignment of specific bus lines for each transfer under consideration of data dependencies in order to avoid parallel write accesses.

3. INTEGRATION INTO THE DESIGN FLOW

At system-level, we use VHDL processes and C programs to describe the behavior of the system. The VHDL and C processes invoke send and receive functions in order to transfer data. An identifier (see Listing 1, T1 and T2) for the value to be transferred allows flexible ordering of the send and their associated receive operations, thus breaking the FIFO-scheme usually inherent to message-based communication. To also allow synchronization, the send and receive functions come in two different formats:
Asynchronous send and receive functions continue immediately after initiation of the transfer, while synchronous transfers block the execution of a process until the transfer has been completed.

Listing 1. Example application

```vhdl
PSrc: process
begin
  A2 := 0;
  wait until Clk = '1';
  A1 := A3 + 3;
  send("PSrc", "PSink", "T1", A1);
  wait until Clk = '1';
  A2 := A1 + A3;
  wait until Clk = '1';
  send("PSrc", "PSink", "T2", A2);
  A1 := 0;
  wait until Clk = '1';
end process;

PSink: process
begin
  B1 := 0;
  receive("PSrc", "PSink", "T2", B2);
  wait until Clk = '1';
  wait until Clk = '1';
  receive("PSrc", "PSink", "T1", B1);
  B3 := B1 - B2;
  wait until Clk = '1';
  wait until Clk = '1';
end process;
```

After validation of the mixed description via co-simulation, we perform control-flow analysis and check by tracing and counting of wait-statements, whether the system contains statically scheduled processes. We consider a process as statically scheduled, if

+ Conditionally executed branches contain the same numbers of wait statements,

+ Only asynchronous send/receive operations occur (transfers that perform unnecessary synchronization are replaced by asynchronous transfers and wait-statements in a preceding step),

+ Loop iteration counts do not depend on external data.

For other processes, different techniques, including sequentializing of transfers and merging of communication channels [Filo et al. 1993; Gasteier et al. 1998], are applied. Communication between processes fulfilling these requirements can be set up by the algorithm presented here. The approach can also be used for software processes with simple control flow such that the timing is predictable on clock cycle level. This, however, will...
usually be impractical if the processor that executes the compiled software supports advanced features that affect the timing, like caching.

The requirement for static scheduling holds mainly for data-flow-oriented applications, like filters. However, more complex applications sometimes also contain components that are suited for the presented approach. This has been demonstrated in the case of an MPEG decoder that has a large computational complexity, memory, and communication bandwidth [Liu 1996].

4. COMMUNICATION SYNTHESIS FOR STATICALLY SCHEDULED SYSTEMS

Throughout this section, we refer to the example presented in Section 3 to illustrate our bus generation approach. A more advanced example will be presented in Section 5.

4.1 Acquisition of Transfer Information

For each transfer, we need an unambiguous identification tag (consisting of the name of the sending and receiving processes and the identifier), the access mode (send/receive), the number of bits transferred, and the exact event time (the clock cycle during which the transfer is executed). This information is extracted during cosimulation and verified by the control-flow analysis step mentioned above. Additional mobility information is added according to the following two steps:

Step 1: Analysis of mobility of each send and receive operation that is limited through (intraprocess) data dependencies.

Step 2: Restriction of mobility ranges due to (interprocess) data dependencies imposed by the send/receive relations.

Figure 1(a) shows the statements executed in our example application (cf. Listing 1) in each clock cycle. From this description, we derive mobility ranges for the values to be transmitted, as shown in Figure 1(b). For example, the value of $A_2$ to be transferred by the second send operation in process $PSrc$ becomes available in clock cycle three, and can therefore not be transmitted until then.
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 \( A2 := A1 + A3 \). When executing the next iteration of process \( \text{PSrc} \), \( A2 \) will again be overridden 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 \( A2 := 0 \). In total, we gain a mobility range from clock cycle three to clock cycle five for \( A2 \), when considering inter-process dependencies as indicated by the solid arrow in Figure 1(b). The mobility ranges of the other operations can be determined analogously.

In the second step, the resulting mobility ranges are restricted further by considering interprocess dependencies, 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 immediately following the beginning of the mobility range of the corresponding send function.

Applying these rules leads to *restricted mobility ranges* as depicted in Figure 1(c). Finally, we determine a transfer mode that can be either one of the following:

- **IMM:** Immediate, the receive operation is executed one clock cycle after the send operation. Considering the VHDL signal semantics, this means that writing to and reading from the bus is done simultaneously, not requiring intermediate storage.

- **SEND:** Send operation, for which the corresponding 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 and 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 I shows the resulting transfer information, assuming that \( T1 \) has a width of 16 bits while \( T2 \) needs 32 bits. The *modified mobility range* \( \phi_{mob}^+ \) contains the first and last clock cycle an operation occupies the bus when executed the first time. Thus, for send operations \( \phi_{mob}^+ \) can be derived from the restricted mobility range by incrementing the interval limits by one while the interval limits of receive operations do not need to be modified.
Both processes need four clock cycles for one iteration. In order to allow more flexible timing schemes, we use two more values, ItNbr and Pause. An iteration is executed ItNbr times before it is suspended for Pause clock cycles.

4.2 Basic Communication Structure

The basic communication structure used for realization of a single cluster is shown in Figure 2. In order to achieve a minimum bus width, multiple accesses can be executed simultaneously, as long as they address different bus lines. In the following, we will use the term bus slice for a set of bus lines addressed by a single access.

RAM components are only inferred if intermediate data storage is required due to disjoint mobility intervals or if a reduction of the total communication costs can be achieved. 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 available data widths.

4.3 Additional Requirements

Scheduling of bus accesses is done 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, which is performed by moving data transfers over wait-statements, four additional requirements have to be taken into account: (a) In case of periodic execution of loops 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; and (d) Accessing RAM usually requires multiple bus cycles, depending on the type of RAM.

4.4 Synthesis Algorithm

The algorithm used to handle the communication synthesis problem as described above is shown in Algorithm 1 and Algorithm 2. It generates a minimum cost bus structure according to the model described in Section 4 for a set of given transfers $T$.
Algorithm 1. Bus generation algorithm

1: $T_{ord} \leftarrow$ rearrange ($T$);
2: $L \leftarrow$ calcScheduleLength ($T_{ord}$);
3: $\Lambda \leftarrow \emptyset; \Lambda_{best} \leftarrow \emptyset; cost_{best} = \infty$;
4: scheduleTrfs ($T_{ord}$, $\Lambda$, $\Lambda_{best}$, $cost_{best}$, $L$);
5: return ($\Lambda$); \{ $\Lambda = \emptyset$ $\Rightarrow$ no feasible solution\}

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 restricted mobility range lengths of the transfers contained in the current transfer set. Although this number decreases, when interprocess data dependencies are considered, it is still a highly intensive task computationally. Computation time can be further reduced by cutting branches of the search tree as soon as possible, which implicitly enumerates all possible schedules.

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 4.

A recursive algorithm is used to create 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 ($cost_{la}$) at any depth-level to the minimum total costs ($cost_{best}$) achieved so far. If $cost_{la}$ exceeds $cost_{best}$, this branch will probably not lead to cost improvement and will be pruned. 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 that can be either executed immediately or with intermediate storage of data, we first process the immediate solutions because these will usually yield lower costs than solutions inferring additional RAM.
Algorithm 2. Function scheduleTrfs (T_{ord}, \Lambda, \Lambda_{best}, cost_{best}, L)

1: \textbf{t = head (T}_{ord};}
2: \textbf{for all } \phi : \phi \in \phi^{+}_{mob}(o^{send}(t)) \cap \phi^{+}_{mob}(o^{rec}(t)) \textbf{ do}
3: \textbf{\lambda \leftarrow findSlice (t);}
4: \textbf{cost_{LA} = calcCostLA (\Lambda \cup \lambda);}
5: \textbf{if cost_{la} < cost_{best} then}
6: \textbf{T}_{ord} \leftarrow T_{ord} \setminus t; \Lambda \leftarrow \Lambda \cup \lambda
7: \textbf{if T}_{ord} = \emptyset \textbf{ then}
8: \textbf{cost_{best} = cost_{la}; \Lambda_{best} = \Lambda}
9: \textbf{else}
10: \textbf{scheduleTrfs (T}_{ord}, \Lambda, \Lambda_{best}, cost_{best}, L);}
11: \textbf{end if}
12: \textbf{end if}
13: \textbf{end for}
14: \textbf{for all } \phi : \phi \in \phi^{+}_{mob}(o^{send}(t)) \cup \phi^{+}_{mob}(o^{rec}(t)) \wedge \phi \notin \phi^{+}_{mob}(o^{send}(t))
\cap \phi^{+}_{mob}(o^{rec}(t)) \textbf{ do}
15: \textbf{for all } RAM \in RAMList \textbf{ do}
16: \text{\{Process like transfer of type IMM but schedule both accesses (send and rec.) according to interfaces of available RAM types and consider additional RAM costs.\}}
17: \textbf{end for}
18: \textbf{end for}

The top level function shown in Algorithm 1 first optimizes the order of transfers by calling the function rearrange(). The function scheduleTrfs() calls itself recursively, thus traversing the search tree. The arguments passed to this function contain the ordered set of transfers T_{ord}, the current mapping of bus slices \Lambda, the best bus mapping achieved so far \Lambda_{best}, the associated costs cost_{best} and the schedule length L.

The first for-loop in function scheduleTrfs(...) (see Algorithm 2) schedules transfers such that they are executed immediately (transfers of type IMM or SEND/REC) while the second loop schedules transfers such that intermediate storage is required (type SEND/REC only). The symbol \phi denotes a clock cycle while o^{send}(t) and o^{rec}(t) refer to the send and receive operation of a transfer t. The function findSlice() searches a free bus slice for scheduling all accesses of a (possibly periodically occurring) transfer. When finding a solution yielding lower cost, the current schedule is stored in \Lambda.

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, and (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 optimum.

5. EXAMPLE APPLICATION

In the following, we will present an example showing some results for the bus generation algorithm. A process DataGen generates data that 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 II.

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

Value FData has to be stored in RAM since the modified mobility range $\phi_{mob}^+$ for receiving this value in process ChkFilter does not overlap the mobility range $\phi_{mob}^+$ 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 bits. The calculated schedule length is 80 clock cycles. Solving this small problem requires 20 milliseconds on a SPARC-20. However, for more complex systems the run-time of the algorithm lies usually within the range up to a few seconds.

<table>
<thead>
<tr>
<th>Mode</th>
<th>SenderID</th>
<th>RecID</th>
<th>Ident</th>
<th>Width</th>
<th>$\phi_{mob}$</th>
<th>Cycle</th>
<th>ItNbr</th>
<th>Pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND</td>
<td>DataGen</td>
<td>Data</td>
<td>10</td>
<td>2–2</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>REC</td>
<td>Filter</td>
<td></td>
<td>2–4</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REC</td>
<td>ChkFilter</td>
<td></td>
<td>2–6</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMM</td>
<td>Filter</td>
<td>EvalChar</td>
<td>8</td>
<td>34</td>
<td>2</td>
<td>8</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>SEND</td>
<td>Filter</td>
<td>FData</td>
<td>16</td>
<td>4–6</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>REC</td>
<td>ChkFilter</td>
<td></td>
<td>5–6</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEND</td>
<td>ChkFilter</td>
<td>Status</td>
<td>8</td>
<td>2–3</td>
<td>8</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>REC</td>
<td>Proc</td>
<td></td>
<td>2–4</td>
<td>8</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMM</td>
<td>P1</td>
<td>P2</td>
<td>ExtTr</td>
<td>8</td>
<td>3–4</td>
<td>16</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
6. CONCLUSION AND FURTHER WORK

This paper presented an algorithm for generation of low cost communication topologies for statically scheduled systems. Buses without arbitration are used for implementation of communication connections.

The next step will be to extend our approach towards a more flexible communication structure. Up to now, our approach cannot handle FIFO-like structures, which require simultaneous read and write operations. This feature can be integrated by introducing Dual-Port-RAM, together with additional mechanisms for generating two separate buses, both connected to that RAM.
REFERENCES


accepted: December 1997