Design Automation for Embedded Systems, vol.2,nr.2, 1997.

LYCOS: the Lyngby Co-Synthesis System

J. MADSEN, J. GRODE, P.V. KNUDSEN, M.E. PETERSEN AND A. HAXTHAUSEN

jan@it.dtu.dk,jnrg@it.dtu.dk,pvk@it.dtu.dk,mep@it.dtu.dk,ah@it.dtu.dk

Department of Information Technology, Technical University of Denmark, DK-2800 Lyngby, Denmark

Editor:

Abstract. This paper describes the LYCOS system, an experimental co-synthesis environment. We present the motivation and philosophy of LYCOS and after an overview of the entire system, the individual parts are described. We use a single CPU, single ASIC target architecture and we describe the techniques we use to estimate metrics concerning hardware, software and communication in this architecture. Finally we present a novel partitioning technique called PACE, which has shown to produce excellent results, and we demonstrate how partitioning is used to do design space exploration.

Keywords: codesign, co-synthesis, hardware/software partitioning, analysis, estimation

1. Introduction

Hardware/software partitioning is often viewed as the synthesis of a target architecture consisting of a single CPU and a single dedicated hardware component (full custom, FPGA, etc.) from an initial system specification, e.g. as in [15]. The partitioning onto such a target architecture is depicted in figure 1.

Even though the single CPU, single ASIC architecture is a special and limited example of a distributed system, the architecture is relevant in many areas such as DSP design, construction of embedded systems, software execution acceleration and hardware emulation and prototyping [43], and it is the most commonly used target architecture for automatic hardware/software partitioning.

In this paper we present the LYCOS (LYngby CO-Synthesis) system which is an experimental co-synthesis environment. In its current version LYCOS may be used for hardware/software partitioning using a target architecture consisting of a single CPU and a single ASIC communicating through memory mapped I/O. In this paper we will focus on how partitioning is done in the LYCOS system and how it is used to do design space exploration.

Consider the hardware/software partitioning as depicted in figure 1 and a set of requirements to be fulfilled by the partitioned system. In order to obtain a feasible partition, that is a partition which fulfills the requirements, we need to know the target architecture, i.e. the CPU on which to run the software, the technology of the dedicated hardware and the interface used for communication between hardware, and software. However, the choice of target architecture will greatly influence the outcome of the partition as each target architecture will have different “best”

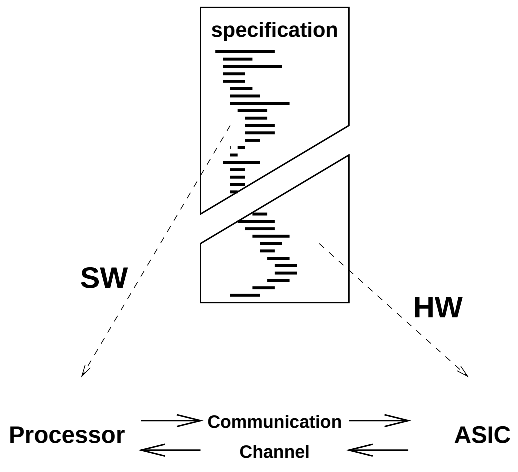


Figure 1. Hardware/software partitioning of a specification onto a target architecture.

partitions. For instance, selecting a *fast* but expensive CPU may lead to little (or no) hardware needed, while selecting a *slow* but cheap CPU may require a large amount of dedicated hardware. Thus, in order to solve the problem efficiently we need a way to explore the design space.

Typically we will have an idea about possible suitable target architectures. These may be selected based on the designer's experience, the desire to reuse predesigned components or the use of third party components. One way to explore the design space is to find the best partition for each possible target architecture and select the best among these.

To find the best partition for a given target architecture, we need a model to represent computation and ways to estimate metrics of software, hardware, and communication.

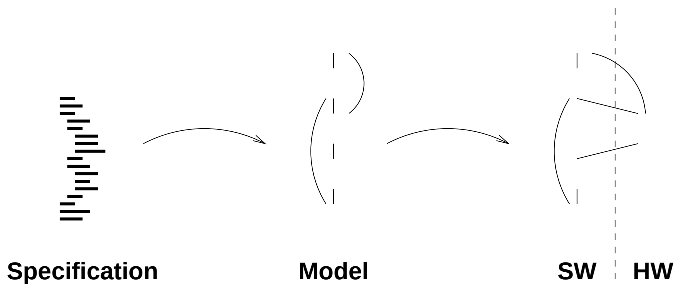


Figure 2. The transformations involved in obtaining a hardware/software partition.

It is important that the model of computation is independent of any particular implementation strategy, that being software or hardware. This will allow for an unbiased design space exploration as well as for translations from various specification languages. One model targeted towards partitioning is based on extracting chunks of computation, called basic scheduling blocks or just blocks. A partition in this model is an enumeration of each block indicating whether it is placed in software or hardware. Figure 2 illustrates the transformations involved in obtaining a partition from the initial specification.

The decision of whether to put a particular block in software or hardware has to be based on an evaluation of the metrics of interest for the entire system. This evaluation can be done in the *physical domain* by actual implementation, e.g. by synthesizing the hardware to a gate netlist on which accurate metrics for area and performance may be obtained, or it can be done in the *model domain* which is less accurate but much faster. For design space exploration we have to do the partitioning and evaluation over and over again, thus, the speed of the evaluation process is a critical issue. In practise, this means that the evaluation has to be done within the model domain, requiring efficient estimation techniques. Figure 3 illustrates how the performance metrics of the system may be obtained from estimators. Other metrics may be obtained in much the same way.

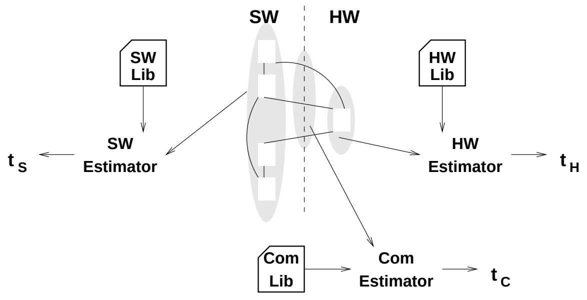


Figure 3. Obtaining performance metrics by the use of estimators.

The paper is organized as follows. In section 2 we outline related work on hardware/software partitioning. Section 3 gives an overview of the LYCOS system. In section 4 we introduce our implementation independent model and present ways to translate from different specification languages into this model. Section 5 presents the system model used for partitioning and how it is analyzed. Section 6 gives an introduction to sections sections 7, 8, and 9 on estimation of metrics for hardware, software, and communication, respectively. In section 10 we present a novel hardware/software partitioning algorithm which forms the basis of automatic hardware/software partitioning in the LYCOS system. Section 11 describes how the LYCOS system may be used to do design space exploration. Finally, we end the paper with a summary and some directions for on-going and future work.

2. Related Work

Several research groups have addressed the problem of co-synthesis and in particular that of hardware/software partitioning. The field of hardware/software partitioning was pioneered by two research groups; COSYMA [14], [15], [27] by Ernst et al. and Vulcan [21], [22] by Gupta and De Micheli.

The COSYMA system performs hardware/software partitioning on an internal representation which is an extended syntax graph. This representation is used for initial analysis such as simulation and profiling. The representation is obtained from a C-like input specification called C^x, which allows for concurrency and timing constraints. The partitioning approach assumes an initial all-software solution and uses a simulated annealing algorithm to move chunks of software code to hardware until the timing constraints are met. COSYMA allows for the use of advanced software structures, such as pointers, in the specification, however, only code which can be implemented directly in hardware is considered for hardware implementation during the partitioning. The algorithm takes communication into account and only variables which need to be transferred are actually considered, i.e. the possibility of local store is exploited. The target architecture for COSYMA is a coprocessor architecture, i.e. one CPU and one ASIC, where the ASIC is used to speedup the CPU. The CPU is RISC based, i.e. heavily pipelined.

Conceptually, the Vulcan system is similar to COSYMA. However, they start by an all-hardware solution specified in HardwareC which fulfills the timing requirements. The specification is translated into an internal graph based representation on which the partitioning is performed. The algorithm, which uses an iterative approach to move operations from hardware to software, takes into account the reduction in communication overhead when neighboring vertices are placed together in either software or hardware. The target architecture consist of a single CPU and one or more ASICs which can all access the memory of the CPU. Whereas the COSYMA system is targeted towards processor speed-up, the Vulcan system is targeted towards designing embedded systems, i.e., the CPU is merely used to reduce the size of the ASICs. Another important difference is that Vulcan can handle multiple processes as hardware and software may run in parallel, whereas COSYMA assumes an interleaved execution of hardware and software.

Two other codesign systems in which hardware/software partitioning is done automatically is SpecSyn [16] and TOSCA [1], [2]. In SpecSyn the input specification is produced in the visual language SpecChart which is based on Statecharts [24]. This is translated into an intermediate system representation called SLIF [47], on which the system analysis and partitioning is performed. SpecSyn supports several partitioning algorithms [16], [48]. [48] presents a combined approach where clustering is used to reduce the number of code blocks to be considered, and a greedy algorithm is used to obtain the partition. The interesting aspect of this approach is that it is able to reach regions in the design space which lies between the regions obtained by fast greedy algorithms and those obtained by the more costly simulated annealing algorithms. In TOSCA the internal representation is based on concurrent hierarchical finite state machines (FSM) which are generated from either standard languages, i.e. C or VHDL, or from higher-level languages such as SpeedChart and Occam. Hardware/software partitioning seems to be done automatically by a clustering algorithm which tries to cluster FSMs based on some closeness criteria. The target architecture for TOSCA is a single standard processor and one or more coprocessors embedded on a single chip.

A number of researchers have focused on algorithm aspects rather than complete systems. Jantsch et al. [30], [31], [32] present a dynamic programming algorithm to solve the partitioning problem of optimizing an existing C-program for speed given a hardware area constraint. The algorithm is derived from the Knapsack Stuffing algorithm [12] and solves (with exponential memory requirements) the partitioning problem for a partitioning model in which blocks can include other blocks and blocks in general therefore cannot be moved to/from hardware independently of each other. A full loop block for example includes the loop body and loop test blocks but all three are considered simultaneously in their model. Kalavade and Lee [35] present a partitioning algorithm which takes communication into account. The partitioning goal is to minimize hardware area given a global execution time constraint. The algorithm has been implemented in the framework of Ptolemy [7] and thus, uses a system level description as input. Another approach using a higher level language as input is presented by Barros et al. [3]. The partitioning algorithm is a two-stage clustering algorithm which selects groups of code based on similarity measures obtained from classification of assignments in the input specification, which is described in UNITY [9].

Codesign systems in which hardware/software partitioning is obtained by user interaction have also been investigated. Among these are POLIS [10], PARTIF [28], and CASTLE [8]. In POLIS analysis and transformations are done on a uniform and formal internal hardware/software representation called Co-design Finite State Machines, CFSMs. Partitioning is done manually by assigning each CFSM to either hardware or software. POLIS will assist the designer by providing estimation tools. POLIS is targeted towards real-time reactive systems, and currently the input specifications are given in Esterel [4] and translated to CFSMs. The target architecture is a system consisting of general purpose processors combined with a few ASICs and possible other components such as DSPs. PARTIF is an interactive partitioning tool which allows the designer to explore different partitionings by applying a small set of transformation and decomposition rules. These rules are applied to a system representation consisting of hierarchical concurrent FSMs. The CASTLE system is another codesign framework. Here the input specification is given in a standard language which can be Verilog, VHDL, or C/C++. This input specification is translated into a common internal representation based on control/dataflow graphs, called SIR, which provides the backbone for all tools. As for POLIS and PARTIF, the partitioning is done manually, but in CASTLE it is based on mappings from a hardware library which is used to specify complex components, including microprocessors.

In the Chinook [11] system the emphasis is on module interface and synchronization. The system is used for real-time reactive controllers initially specified in Verilog. Chinook does not provide automatic hardware/software partitioning, but leaves it to the designer, nor does it provide code generation tools for the target processors, but uses standard C compilers. However, Chinook does synthesize the hardware and software needed for inter process communication which is a difficult task as the different components may not initially fit very well together.

Research in distributed system co-synthesis has recently recieved a lot of attention, for instance by Wolf and Ti-Yen [49], [50] and by Jerraya et al. [29].

From the discussion on related work it becomes evident that there are two major directions for hardware/software partitioning methods; Automatic partitioning which in almost all cases means a restricted target architecture, i.e., one CPU and one or maybe a few ASICs, and manual partitioning which typically allows for more advanced target architectures. Also, when having advanced target architectures, synchronization and communication between different components becomes much more difficult and very important (as for instance in the Chinook system). This is an aspect which becomes even more important when considering distributed system co-synthesis.

3. Overview of the LYCOS System

In this section we describe the main ideas and motivations for building the LYCOS system and give an outline of how the system has been implemented.

One of the main ideas of the LYCOS system is to have a system which supports an easy inclusion of new design tools and algorithms, as well as new design methods, e.g. the sequence in which tools have to be applied in order to obtain a solution. Thus, a key issue for LYCOS has been to be able to test new ideas and algorithms not as separate entities but as part of a complete design flow. This has to a large extent required the development of our own tools rather than trying to integrate existing tools, that being commercial or university tools. However, from the register transfer level and down to the final layout we are relying on existing commercial design tools.

Figure 4 gives an overview of the LYCOS system. LYCOS is built as a suite of tools centered around an implementation independent model of computation, called Quenya, which is based on communicating control/data flow graphs.

The information which must be communicated between different tools in LYCOS basically consists of two parts: The central functionality of a design (i.e. the behavior derived from the input specification) and design hints obtained during synthesis (e.g. partitioning or scheduling information, profiling, and performance). In order to achieve the necessary flexibility to support the requirements of different kinds of synthesis tools, while preserving the semantic integrity of the description, this dichotomy has been reflected in the design of Quenya.

The functional behavior of a design is represented through a hierarchical network with strictly defined semantics. The network consists of a number of functional

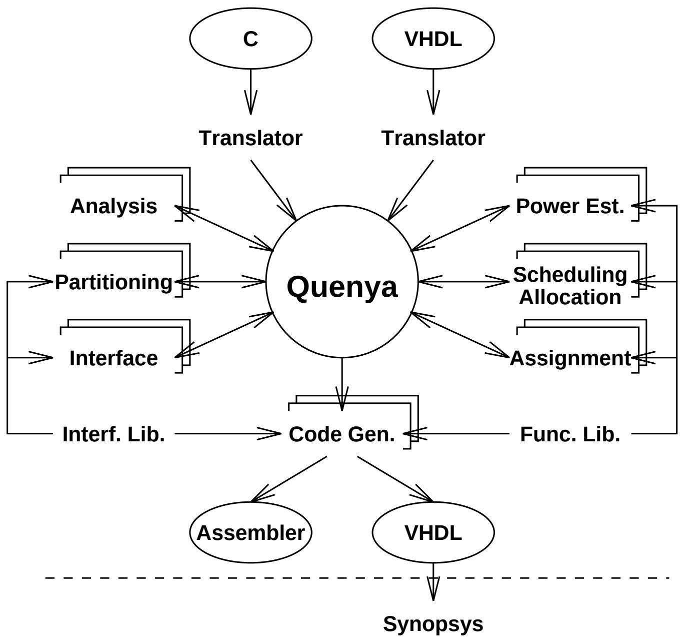


Figure 4. Overview of the LYCOS system.

units communicating by asynchronous protocols, with the communication channels represented as shared variables. Thus, Quenya directly allows design partitioning to be represented, and supports representation of the system's environment in a uniform manner. The behavior of an individual functional unit is represented by a control/data flow graph (CDFG). The basic CDFG provides a uniform representation of both data and control flow, utilizing lossless and self-timed communication of tokens, i.e. points of execution with associated data. This basic model for functional behavior has been extended to support inter-domain communication, allowing the communication between functional units to be modeled.

Different design problems require different combinations of tools and parameter settings in order to obtain a suitable solution, and as the right order for a given design problem is not evident, exploration is needed. Quenya has been designed with this in mind by keeping tool specific information as annotations to the network and graphs

In this paper we focus on how the LYCOS system is used to do automatic hardware/software partitioning. Though Quenya is able to represent a distributed system, automatic synthesis is currently targeted towards a coprocessor based target architecture, i.e. a system consisting of a single CPU and a single ASIC. In the following sections we first present the design flow to obtain a partition from an initial specification in C or VHDL. We then describe how LYCOS may be used to do design space exploration.

4. Application Descriptions

The application to be partitioned is externally described by a specification written in some specification language.

As we would like the LYCOS system to support several specification languages (see section 4.1), one of the important design decisions for LYCOS was to make the system independent of the choice of specification language. This has been achieved by basing the system on a common internal representation which serves as interface between the specification languages and the partitioning tool. For the common internal representation we use Quenya CDFGs, in the following refered to as CDFGs or graphs.

Hence, specifications must first be translated into this internal CDFG representation before partitioning can be done.

In the following subsections we present the internal CDFG representation, the supported specification languages and their translation into CDFGs.

4.1. Specification Languages

LYCOS currently supports specifications written in translatable subsets of the programming language C and the hardware specification language VHDL.

Due to the limits in the expressive power of CDFGs not all language constructs can be translated into CDFGs. For instance, pointers cannot be translated, as CDFGs cannot handle dynamic storage allocation. We are currently investigating possibilities for extending the CDFG format such that more language constructs can be translated.

The reason for supporting C as a specification language is that many existing software applications are written in C and our basic target architecture is well suited for software speedup. In order to be able to specify applications with multiple processes without extending C with some hardware functionality, we chose VHDL as a specification language too. VHDL has expressive power in this direction, but of course, other HDL languages, such as Verilog, could have been chosen.

In the future we would also like to support formal specification languages like RSL [45] and Synchronized Transitions [46]. Such languages together with their associated methods and support tools have shown to be useful for specifying systems. One of the reasons for this is that they provide a higher level of abstraction than conventional languages like C and VHDL making it possible to specify what the properties of the system should be without giving implementation details like particular data type representations and algorithms. Furthermore, the formal basis of these languages makes it possible to formally prove properties of specifications and thereby increase the reliability of the specifications.

4.1.1. Specification for the Straight example

In figure 5, a VHDL specification of an application called Straight is shown. It has proven to be a good example to use to show different aspects of our tool suite and algorithms as well as to highlight interesting aspects of partitioning and co-synthesis. We will use this example throughout the paper to illustrate our approach. More complex examples containing loops and conditionals are considered in section 11.

```
entity straight is
 port (clk, load : in BOOLEAN;
    a_i, dz_i, z_i, u_i, y_i : in INTEGER;
    u_o, y_o, z_o : out INTEGER);
end straight;

architecture behavioral of straight is
begin
 process
   variable a, dz, z, u, y : INTEGER;
   variable z1, u1, y1, z2, u2, y2 : INTEGER;
   variable z3, u3, y3, z4, u4, y4 : INTEGER;
   variable z5, u5, y5, z6, u6, y6 : INTEGER;
   variable z7, u7, y7, z8, u8, y8 : INTEGER;
   variable z9, u9, y9 : INTEGER;

procedure Block0(a, dz, z, u, y : in INTEGER;
         u_o, y_o, z_o : out INTEGER) is
begin
 z_o := z; u_o := u; y_o := y;
end;

procedure Block1(a, dz, z, u, y : in INTEGER;
         u_o, y_o, z_o : out INTEGER) is
begin
 z_o := z + dz;
 u_o := u - (3*z*u*dz) - (3*y*dz);
 y_o := y + (u*dz);
end;

procedure Block2(a, dz, z, u, y : in INTEGER;
         u_o, y_o, z_o : out INTEGER) is
variable
   z1, u1, y1 : INTEGER;
begin
 z1 := z + dz;
 u1 := u - (3*z*u*dz) - (3*y*dz);
 y1 := y + (u*dz);
 z_o := z1 + dz;
 u_o := u1 - (3*z1*u1*dz) - (3*y1*dz);
 y_o := y1 + (u1*dz);
end;

procedure Block3(a, dz, z, u, y : in INTEGER;
         u_o, y_o, z_o : out INTEGER) is
 variable z1, u1, y1 : INTEGER;
 variable z2, u2, y2 : INTEGER;
begin
 z1 := z + dz;
 u1 := u - (3*z*u*dz) - (3*y*dz);
 y1 := y + (u*dz);
 z2 := z1 + dz;
 u2 := u1 - (3*z1*u1*dz) - (3*y1*dz);
 y2 := y1 + (u1*dz);
 z_o := z2 + dz;
 u_o := u2 - (3*z2*u2*dz) - (3*y2*dz);
 y_o := y2 + (u2*dz);
end;

 begin
  -- wait for the rising clock edge
  if not clk then
   wait until clk;
  end if;

  -- Load the input values
  a := a_i;
  dz := dz_i;
  z := z_i;
  y := y_i;
  u := u_i;

  Block1(a, dz, z, u, y, u1, y1, z1);

  Block0(a, dz, z1, u1, y1, u2, y2, z2);

  Block1(a, dz, z2, u2, y2, u3, y3, z3);

  Block2(a, dz, u3, y3, z3, u4, y4, z4);

  Block2(a, dz, u4, y4, z4, u5, y5, z5);

  Block3(a, dz, u4, y4, z4, u6, y6, z6);

  -- Store the output values
  u_o < = u6;
  y_o < = y6;
  z_o < = z6;

  if clk then
   wait until not clk;
  end if;
 end process;
end behavioral;
```

Figure 5. VHDL specification for the Straight example

The bodies of the functions `Block1`, `Block2`, and `Block3` is taken from the HAL example [44]. In `Block2`, the code has been duplicated with data dependencies made from first to second copy of the HAL example. In `Block3`, it has been triplicated. Note that the functionality of `Block0` is merely a transfer of values.

4.2. The Internal CDFG Representation

This section provides a short introduction to the Quenya CDFGs. A more detailed description of the CDFGs is given in [6] which extends the CDFGs defined in [13], and the computational semantics of the CDFGs has been formalized in [41]. The formalization ensures that the meaning of CDFGs is unambiguous and makes it possible to formally reason about CDFGs [5] (e.g. to prove that a CDFG has the desired input-output behaviour or that a CDFG transformation is correct).

The purpose of CDFGs is to represent computations. An example of a CDFG is given in figure 6.

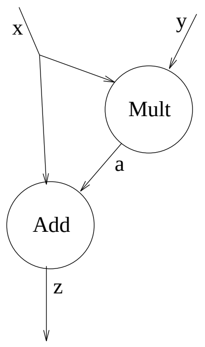


Figure 6. A CDFG representing the computation `z := x + x * y`

A CDFG is a hierarchical directed hypergraph consisting of nodes and edges. The semantics is based on a token passing mechanism, similar to colored Petri nets [33], [36]. The edges are entities on which tokens (i.e. values) can flow between nodes. Nodes can remove tokens from their input edges and place tokens on their output edges according to certain *firing* rules. There are different kinds of nodes. The nodes shown in figure 6 are all *infix nodes*, which have two input edges, one output edge and an associated infix operator. When an infix node, *op*, has tokens, say *v1* and *v2*, on its input edges and no token on its output edge, it can fire by placing the token, *v1 op v2*, on its output edge, and removing *v1* and *v2* from its input edges. Other kinds of nodes are prefix nodes, constant nodes, control nodes to express conditionals, iteration nodes to express loops, void nodes to absorb tokens from edges, etc. For more details on these nodes, see [6], [41]. A graph is *executed* by placing tokens on its input edges and letting the nodes fire until no more firing rules can be satisfied.

It is possible for a collection of CDFGs to communicate with each other through shared variables. Special interface nodes are used for this: Import and export nodes for sampling and updating the contents of shared variables, respectively, and wait nodes for synchronization to global events (i.e. certain contents of shared variables). Figur 7 gives an example of two CDFGs, Graph1 and Graph2, which communicate through two shared variables, `s` and `ok`. Graph1 has two export nodes, E1 and E2, which can update `s` and `ok`, respectively, with the value of a token on the `x` edge and `b` edge, respectively. Graph2 has a wait node, W1, which requires `ok` to contain the value `true` in order to fire. In addition Graph2 has an import node, I1, which can sample the value of `s` and place it on the `y` edge. A common feature of the interface nodes is that they all require a token on their vertical input edge in order to fire, and that they generate a token on their vertical output edge after firing. This feature has been used to ensure a certain sequencing of the events. For instance, I1 must wait sampling `s` until W1 has fired.

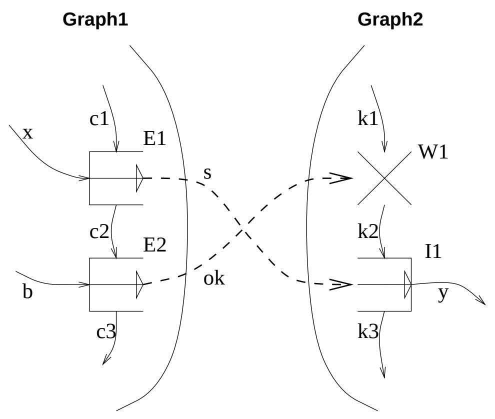


Figure 7. Two CDFGs communicating through shared variables.

4.3. Translation

Having formulated a specification in one of the supported specification languages, the specification should be translated into a CDFG which represents the same input/output behaviour as the specification.

Below we will explain the translation of the most important VHDL language constructs. The translation of the corresponding C language constructs is the same, unless otherwise stated. For a more detailed explanation of the translation from C and VHDL, see [25], [26], and [6], respectively.

4.3.1. Assignments and expressions

In section 4.2 we saw that the graph in figure 6 represents the same computation as the assignment `z := x + x * y`. This example gives an idea of how assignments and expressions are translated.

As should be obvious from the example, there is a close correspondance between variables in a specification and edges in the graph into which it is translated. At each point of the specification any variable has a *corresponding edge.* Each time an assignment is made to the variable, it gets another corresponding edge. In this way several edges may correspond to the same variable. For instance, the assignment `x := x + x * y` is translated into the graph shown in figure 8, where `x1` is the corresponding edge of `x` before the assignment and `x2` is the corresponding edge of `x` after the assignment.

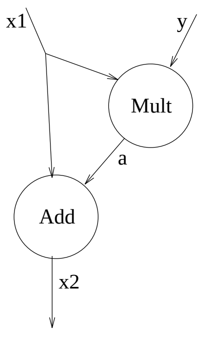


Figure 8. A CDFG representing `x := x + x * y`

4.3.2. Conditionals

An if statement, **if** `e` **then** `s1` **else** `s2` **end if**, is translated into a graph whose shape is shown in figure 9. The graph is composed of the graphs of `e`, `s1` and `s2`, and a number of branch nodes (those nodes marked with a B inside) and merge nodes (those marked with an M inside). The branch and merge nodes all have as their control input edges the output edge *b* of the graph of the test expression `e`. The graph is executed as follows. First Graph(e) will be executed leading to a token on the *b* edge. Depending on whether this token is *true* or *false*, the branch nodes will move the tokens of their vertical input edges to the input edges of the sub-graph of `s1` or `s2`, respectively. After execution of the selected sub-graph, the merge nodes will move the tokens of the output edges of the selected sub-graph to the output edges of the merge nodes.

4.3.3. Loops

A while statement, **while** `e` **loop** `s` **end loop**, is translated into a graph whose shape is shown in figure 10. The graph is composed of the graphs of `e` and `s`, and a number of entry nodes (those marked with an En inside) and exit nodes (marked with an Ex inside). The entry and exit nodes all have as their control input edges the output edge, *b*, of Graph(e). The graph is executed as follows. First the entry nodes will move the tokens of their right vertical input edges to the input edges of Graph(e) which will then be executed. After execution of Graph(e) the *b* edge will contain a new token. If it is false, the exit nodes will move the tokens of the output edges of Graph(e) to the rightmost output edges of the exit nodes and the execution of the graph is thereby finished. Otherwise the exit nodes will move the

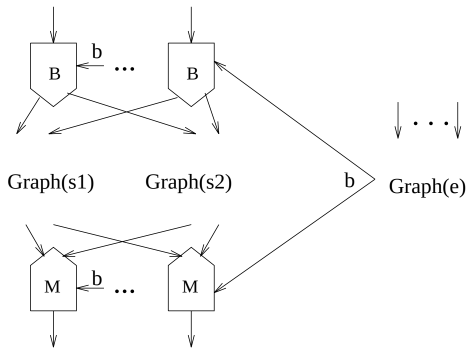


Figure 9. Translation of if statements, **if e then s1 else s2 end if**

tokens of the output edges to the input edges of Graph(s) and after the execution of that, the entry nodes will move the tokens of the output edges of Graph(s) to the input edges of Graph(e) and the execution will continue as before. For loops and repeat-until loops are translated similarly.

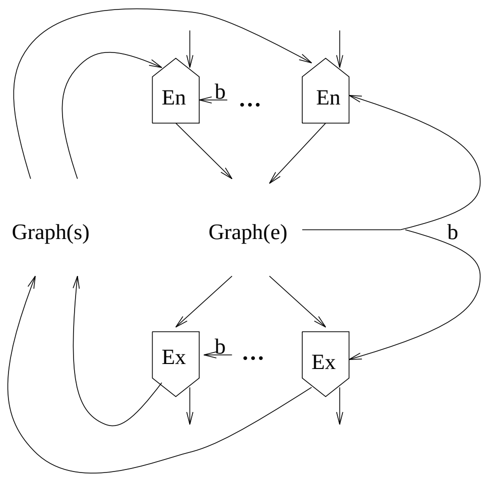


Figure 10. Translation of while statements, **while e loop s end loop**

4.3.4. Functions and procedures

The body of a function or procedure is translated into a seperate graph. A function or procedure call induces a node in the graph that represents the call. For each input parameter, an input edge is connected to the node, and for each output variable an output edge is connected. See section 4.3.6 for an example.

4.3.5. Communication and synchronization

In VHDL it is possible to specify a set of communicating processes that use the wait statement for synchronization. The wait statement is translated into a wait node and communication expressions are translated into import and export nodes.

4.3.6. Translation for the Straight example

Translation of the specification shown in figure 5 results in a hierarchical graph. In figure 11, the leftmost graph shows the top level of the graph corresponding to the `process` declaration (the implicit infinite loop of a VHDL process has been left out for convenience). The rightmost graph shows the calculations of one of the functions, `block1`.

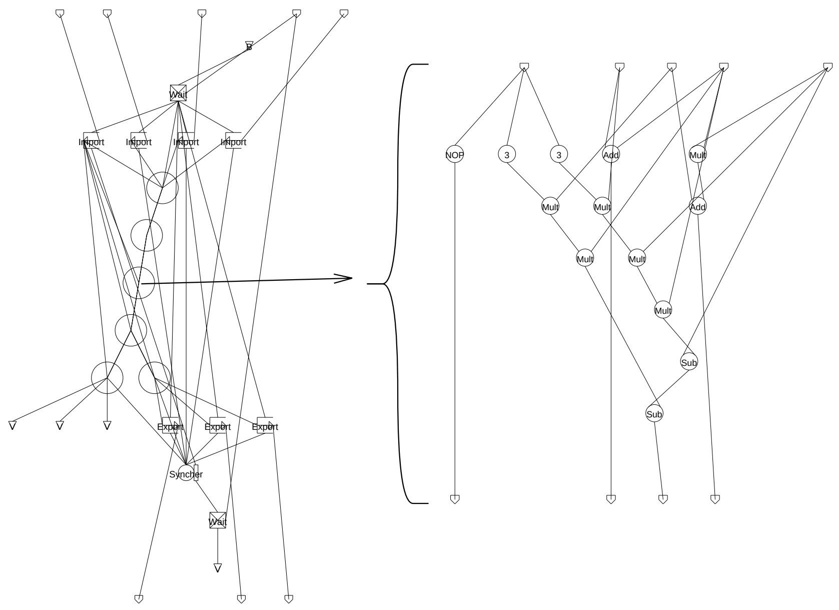


Figure 11. Graph with two levels of hierarchy for the Straight example.

5. System Modeling and Analysis

This section presents the system model used by the partitioning algorithm, and describes how it is obtained from the internal representation.

5.1. The Control Flow Graph Interface Format

Since we handle complex applications, the Quenya CDFGs tend to grow in complexity. Because of this, an interface to the internal representation has been defined.

The basic idea in the Control Flow Graph Interface Format (ConGIF) is to focus on the control flow of the application. It is however important to note that no information is lost in ConGIF. ConGIF CDFGs are defined as follows:

Definition. A ConGIF CDFG (N, E) is a set of nodes and directed edges where an edge $e_{i,j} = (n_i, n_j)$ from $n_i \in N$ to $n_j \in N$, $i \neq j$, indicates that n_j depends on n_i because of data dependencies and/or control dependencies. A node n is defined as:

$$\begin{aligned} n &= \text{DFG} \mid \text{Cond} \mid \text{Loop} \mid \text{FU} \mid \text{Wait} \\ \text{Cond} &= (\text{Branch1}, \text{Branch2}) \\ \text{Loop} &= (\text{Test}, \text{Body}) \\ \text{Branch1} &= \text{ConGIF CDFG} \\ \text{Branch2} &= \text{ConGIF CDFG} \\ \text{Test} &= \text{ConGIF CDFG} \\ \text{Body} &= \text{ConGIF CDFG} \\ \text{FU} &= \text{ConGIF CDFG} \end{aligned}$$

where a DFG is a pure dataflow graph without control structures, FU represents a function or procedure call, Wait is used for synchronization with the environment, Branch1 and Branch2 are the CDFGs to be executed in the “true” and “false” branch case of a conditional Cond, respectively, and Test and Body are the test and body CDFGs of a Loop.

An important limitation of the derivation of the ConGIF CDFG from the Quenya CDFG is that control flow is handled in an “as soon as possible” manner. This means that the control flow of the ConGIF CDFG basically follows the sequential way of writing the specification of the application. As a result the full control flow parallelism within the application is not expressed directly in the graph. It is, however, not lost as it can be deduced by a data dependency analysis on the control flow level [19].

5.2. Derivation of Basic Scheduling Blocks

In order to be able to partition the application, the corresponding ConGIF CDFG must first be divided into Basic Scheduling Blocks (BSBs) that may be moved between hardware and software. For each node in the ConGIF CDFG, a BSB is created as shown in figure 12. Each BSB can have child BSBs which are shown

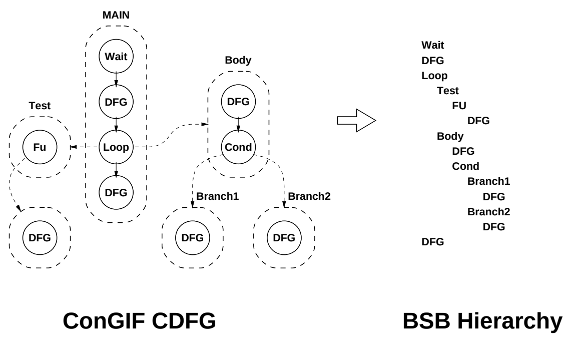


Figure 12. The BSB hierarchy and its correspondence with the ConGIF CDFG.

indented under the BSB. In this way a *BSB hierarchy* which reflects the hierarchy of the application is obtained. With each BSB we associate information which is used by the partitioning algorithm to determine whether it should be placed in hardware or software:

Definition. For a BSB, B_i, the function $\mathit{info}(B_i)$ returns the associated information:

$$\mathit{info}(B_i) \;=\; \langle a_{s,i}, t_{s,i}, a_{h,i}, t_{h,i}, r_i, w_i \rangle$$

where $a_{s,i}$ and $t_{s,i}$ are the area (code size) and execution time of B_i when placed in software, $a_{h,i}$ and $t_{h,i}$ are the area and execution time of B_i when placed in hardware and r_i and w_i contain the read-set and write-set variables of B_i. The read-set of a BSB contains the variables that are read by the BSB. The write-set of a BSB contains the variables that are written by the BSB.

In order to be able to control the granularity (number and sizes) of the BSBs which are considered by the partitioning algorithm, parent BSBs can be collapsed as to appear as single BSBs instead of the child BSBs they are composed of. This is illustrated in figure 13.

The partitioning algorithm only considers *leaf BSBs* which are BSBs which have no children. The leaf BSBs are marked with a dot in the figure. When BSBs are collapsed, the number of leaf BSBs decreases. In this way, the run-time of partitioning algorithms which depends on the number of BSBs can be reduced.

As all leaf BSBs together make up the application, we can now define the application in terms of leaf BSBs:

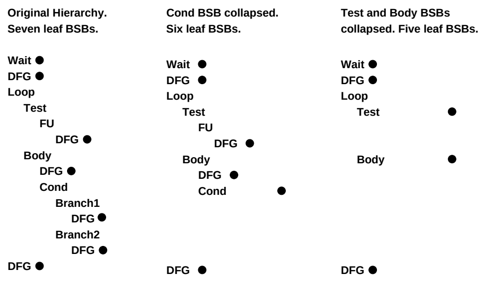

Original Hierarchy. Seven leaf BSBs.
Cond BSB collapsed. Six leaf BSBs.
Test and Body BSBs collapsed. Five leaf BSBs.
Wait
DFG
Loop
Test
FU
DFG
Body
DFG
Cond
Branch1
DFG
Branch2
DFG
DFG

Figure 13. Adjusting BSB granularity by hierarchical collapsing.

Definition. An application is a sequence S of n leaf BSBs, i.e.

$$S = \langle B_1, B_2, \ldots, B_n \rangle$$

where B_i denotes BSB number i and n is the total number of leaf BSBs.

In order to estimate performance, it is necessary to know how many times each BSB is executed for typical input data. This information is obtained from profiling, see section 5.3. At this point it is convenient to define two global functions which return profiling information for individual BSBs and individual variables:

Definition. The function $pc(B_i)$ returns the number of times a BSB has been executed in a profiling run ("pc" is short for "profiling count").

Definition. The function $ac(v)$ (short for "access count") takes any variable v in the application and returns the number of times the variable is accessed.

5.3. Profiling

The purpose of profiling is to determine the values returned by the functions *pc* and *ac* for a given application and a given set of input data. This will be referred to as the profile.

Since we allow applications with data dependent loops, conditionals, and synchronizations, it is in general not possible to determine a static execution profile for a given application. However, from a partitioning point of view it is important to have some information about time critical parts of the application, since these might have to be moved into hardware in order to meet timing constraints. If the input data used for profiling are carefully selected, the profile will expose potentially time critical parts of the application.

It should be emphasized that the profile is a *hint* to the partitioning tools. It is not a worst case analysis and cannot be used when considering applications with hard timing constraints. Also the profile could differ widely for different sets of input data.

Profiling is performed on the Quenya representation and is therefore independent of the final implementation. Instead of writing a dedicated CDFG simulator, the profile is obtained by translating the CDFG to C++. During the translation, code is added to collect the execution count during execution. The translation is a simple linearization of the CDFG representation. Multiple processes are supported by executing one process at a time until it is suspended, in a round-robin fashion.

It is necessary to supply a test environment. This may either be written directly in C++ or supplied as a CDFG (translated from either C or VHDL). In the latter case the test environment should be translated to C++. Writing the test environment requires a minimal amount of work since it should merely implement the communication protocol.

Executing the C++ program is much faster than running a CDFG simulator and allows us to use larger sets of input data. Once created, the profile is annotated back to the CDFG. Also, apart from providing valuable information about the execution of the CDFG the execution of the C++ program may be used for a simple validation of the functionality.

The profile takes into account the hierarchy of the application. This means that for each function the profiler will keep separate counts for different callers. Therefore, to get the total number of times a function has been called it is necessary to sum the individual counts.

5.3.1. Profiling the Straight Example

The example in section 4.3.6 contains no loops or branches so the individual counts will be equal for all parts of the application. However, since `Block1` and `Block2` are called twice, each will have two counts, therefore the *total* count for these blocks will be twice the counts for the other blocks.

6. Estimation aspects

Just as it is necessary to be able to operate with different tradeoffs between accuracy and execution time for the partitioning algorithms, it is also necessary to be able to operate on different levels of estimation accuracy. Fast but less accurate estimates must be used in the design space exploration phase, while more accurate and time consuming estimation methods should be used in the later phases.

The purpose of estimation is to achieve as accurate measures as needed of the outcome of the co-synthesis process. Typical measures are software and hardware execution time, communication time, software object code size (including size of communication routines) and hardware area (including communication aggregate area). The most accurate measures are of course obtained by actually performing the co-synthesis process (from which the software- and hardware area measures can be determined) and then executing the result on a given target (which results in the dynamic hardware, software and communication time measures), but this is a very time consuming process, and is clearly infeasible when we consider that the partitioning algorithms require estimates for all considered partitions for an architecture and on top of that are executed repeatedly for a lot of different target configurations.

So how do we decrease estimation time (on the expense of estimation accuracy) in the best way? First we must note that the estimation problem can be divided into two subproblems, namely

1. Modeling of the hardware- and software compilation process which results in static area measures.
2. Modeling of the execution process which results in dynamic performance measures. This ultimately includes modeling of performance characteristics of communication protocols, caches, pipelines, prefetch queues, snooping, waitstates, etc.

Both kinds of models have to be as accurate as possible and to be consistent with each other. These objectives are best satisfied if we know how the software- and hardware compiler are constructed. The most accurate knowledge is obtained if we do not use third party compilers, but build them ourself instead. This illustrates the importance of *tool integration* which we aim at achieving in the LYCOS system by constructing the tools ourselves in a way that let them easily communicate estimates and results to each other.

When using external hardware/software compilers it is very difficult to model them exactly, and this may result in estimates which differ considerably from the outcome of these compilers. This means that the partitioning algorithms make decisions on a wrong basis, and thereby result in worse target configurations and functional partitions and in constraints that may not be satisfied.

Section 5 has shown how we obtain profiling estimates and sections 7 to 9 describe in detail how hardware, software and communication estimates are obtained. Currently only one fast level of estimate accuracy is supported. It utilizes the same hardware library and hardware scheduler that will ultimately be used for hardware compilation but only a simple generalized software compiler model. As we continue development of hardware- and software cross compilers, we intend to support more accurate levels of estimates which will enable us to estimate for example global optimizations over BSB boundaries that these compilers may perform. With regard to dynamic execution estimation, we currently only model simple non-pipelined software processors (but do support the use of pipelined hardware modules), and disregard complex matters as caches, etc.

7. Hardware Modeling and Estimation

In order to be able to estimate the hardware area of the final result from hardware synthesis, we model the hardware target architecture in a very general manner, using the Architectural Construction Environment, ACE, as described in [23]. This environment is used to describe the hardware components of our target architecture, and it has the following key features:

- A hardware component is described in an `Architecture` and a `Functional Module`.
- The `Architecture` is a "black box" that describes the interface of the component.
- The `Architecture` has several (optional) attributes, where the most important are estimated area of the hardware component and the minimum cycle time for correct functional operation. The estimates are generated by the library designer.
- The `Functional Module` describes the functionality of the component. The module describes the operations provided by the component, and the communication protocols of these operations (Protocol Flow Graphs, PFGs), that is, the way to interface with the component. Also storage devices within the component that are usable from outside of the component are listed in the `Functional Module`.
- The `Functional Module` allows for descriptions of combinatorial and pipelined components.

The representation contains several additional constructs.

```
(FIXED-ARCHITECTURE mul-ser
  (TYPES
    (INTEGER-TYPE int 0x0000 0xFFFF)
  )
  (PORTS
    (IN int in1)
    (IN int in2)
    (OUT int out1)
  )
  (AREA-ESTIMATE 103 103 103)
  (MINIMUM-CYCLE-TIME 14.8)
)
```

```
(FUNCTIONAL-MODULE mul-ser
  (ARCHITECTURE mul-ser)
  (OPERATIONS
    (OPERATION multiply ((IN int a) (IN int b)
                         (RETURN int c))
      (LATENCY 15)
      (PFG
        (TRANSFER (mul-ser (in a) (in b)
                           (out c)))
      )
    )
  )
  (STORAGE)
  (INSTRUCTIONS
    (INSTRUCTION mul-ser ((IN int a) (IN int b)
                          (OUT int c))
      (DELAY 15)
      (TRANSFER
        (CYCLE 0
          (INPUT  a in1  (DELAY 0))
          (INPUT  b in2  (DELAY 0))
        )
        (CYCLE 15
          (OUTPUT c out1 (DELAY 14.8))
        )
      )
    )
  )
)
```

Figure 14. The `Architecture` and `Functional Module` of a serial multiplier described in ACE

In figure 14 a serial multiplier is described in ACE. The `Architecture` defines the type `int`, which is the type of the three ports of the multiplier (two inputs, one output). Also, the estimated area (minimum, typical and maximum) is listed together with the minimum cycle time for correct operation of the multiplier. In the `Functional Module`, the operations of the component are listed, in this case only one, and for each operation, the `PFG` dictates how to interface to the operation. Each `Functional Module` is internally composed of a set of instructions to which the operations refer (one operation can refer to several instructions). Each instruction is composed as a list of transfers that describes the instruction in a *cycle true* manner.

The multiplier shown in figure 14 is a "pipelined" component with equal latency (of the operation `multiply`) and delay (of the instruction `mul-ser`). In such cases we refer to the component as multicycled (another case of multicycled components is when the operation of a combinatorial component is stretched over several clock cycles).

In cycle 0, the values on input-ports `in1` and `in2` of the architecture are transferred (by the `PFG` of the operation `multiply`) to the instruction `mul-ser` through the parameters `a` and `b` of this instruction. After a delay of 15 cycles, the result is returned through the output parameter `c` to the output-port `out1`.

The component would have been pipelined if the latency was less than the delay (the operation could be re-invoked before the current computation(s) has produced its output). If the delay is zero, which implies the latency to be zero too, the component is combinatorial.

ACE may also be used to describe communication channels [42], and we are currently investigating the possibility of describing software components like microprocessors and ASIPs in the representation.

7.1. Hardware Area Estimation

Using the ACE models of the components in the hardware part of a given target architecture, the total hardware area for a given implementation can be estimated.

A common way of estimating the hardware area of a BSB is to estimate how much area a full hardware implementation of the BSB will occupy [31], [35]. This includes hardware to do the calculations of the BSB and hardware to control the sequencing of these calculations. If the total chip area is divided into a *datapath area* and a *controller area*, each BSB moved to hardware may be viewed as occupying a part of the datapath and a part of the controller. Figure 15a shows this model when one BSB has been moved to hardware.

When more than one BSB are moved to hardware they may share hardware modules as they do not execute in parallel. Hence, approaches which estimate area as the summation of datapath and control areas for all hardware BSBs will probably overestimate the total area. This problem is depicted in figure 15b where the area of the datapath is *not* equal to the sum of the individual BSB datapaths.

In our approach, the datapath area, a_{dp}, is the area of a set of preallocated hardware resources in the datapath as illustrated in figure 15c, i.e.

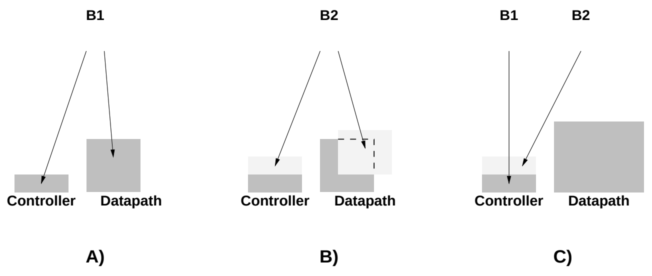


Figure 15. BSB area estimation which accounts for hardware sharing: a) Controller- and datapath area for a single BSB, b) When sharing hardware, the total area for multiple BSBs is less than the summation of the individual areas, c) Variable controller area and fixed datapath area for multiple BSBs with hardware sharing.

$$a_{dp} \;= \sum_{r \in \mathcal{R}} a_{dp,r}$$

where $\mathcal{R}$ is the set of preallocated resources and $a_{dp,r}$ is the area estimate of the individual resources (components). The area estimate for each component is obtained from the hardware target architecture described in ACE.

If the total chip area is denoted a_{total} we can write the area left for the BSB controllers, a_c, as:

$$a_c \;= a_{total} - a_{dp} = a_{total} - \sum_{r \in \mathcal{R}} a_{dp,r}$$

The BSBs share the preallocated resources, and therefore we are only concerned with the area cost of implementing the BSB controllers. Thus, the hardware area of a BSB is estimated as the hardware area of the corresponding controller and is for a BSB, B_i, denoted $a_{h,i}$ (c.f. section 5.2). The hardware estimate of a BSB (i.e. a BSB controller) will depend on the number of timesteps required for executing the BSB [37].

7.2. Hardware Execution Time Estimation

The hardware execution time for a BSB is determined by the scheduler in our tool suite [18]. The scheduler is based on dynamic list scheduling but with the enhancement that the tool handles the control structure of the applications as well. The overall algorithm **ScheduleCDFG** is shown in figure 16. The algorithm schedules the ConGIF CDFG $\mathcal{G}$ using the resources (hardware datapath components) $\mathcal{R}$. The algorithm schedules the control-flow of the algorithm, by following the order of control-structures induced by ConGIF. This means that no control-structures are scheduled in parallel (e.g. a loop and a conditional), hence, the parallelism within the application is not utilized to its full extend. Scheduling control-structures (i.e. BSBs) in parallel is however of major concern when trying to speed up an application, and the problem has been dealt with in [19]. This scheduling algorithm has, however, not been incorporated in LYCOS yet, since the partitioning algorithm at this point only considers BSBs that execute in mutual exclusion.

The reason that functions (FU's) are only scheduled once is that they can only have one schedule each, at least for a pure hardware solution. The resources are shared between the different parts of the CDFG due to mutual exclusion.

```
ScheduleCDFG($\mathcal{G}$, $\mathcal{R}$) ≡
{
 for n = first node to last node in $\mathcal{G}$ do {
  case n {
   DFG : ScheduleDFG(n,$\mathcal{R}$)
   Cond : {
    ScheduleCDFG(Branch1,$\mathcal{R}$);
    ScheduleCDFG(Branch2,$\mathcal{R}$)
   }
   Loop : {
    ScheduleCDFG(Test,$\mathcal{R}$);
    ScheduleCDFG(Body,$\mathcal{R}$)
   }
   FU : {
    if FU has not been scheduled yet then {
     ScheduleCDFG(FU,$\mathcal{R}$)
    }
   }
   Wait : {
    Schedule n in current control-step
   }
  }
 }
}
```

```
ScheduleDFG($\mathcal{D}$, $\mathcal{R}$) ≡
{
 Calculate ASAP values for all nodes in $\mathcal{D}$.
 $\mathcal{L}$ = list of nodes ready to be scheduled
 while not done {
  for_all n in $\mathcal{L}$ do {
   Urgency(n) = ASAP(n) - current control step
  }
  $\mathcal{R}_t = \mathcal{R}$;
  Sort($\mathcal{L}$, Urgency);
  for m = first in $\mathcal{L}$ to last in $\mathcal{L}$ do {
   if Op(m) $\in \mathcal{R}_t$ {
    Schedule m in current c step;
    $\mathcal{L} = \mathcal{L} \setminus m$;
    $\mathcal{R}_t = \mathcal{R}_t \setminus$ Op(m)
   }
  }
  Add new ready nodes to $\mathcal{L}$;
  Increase current c step
 }
}
```

Figure 16. Basic scheduling algorithms

When scheduling the data flow using the function **ScheduleDFG**, which is based on dynamic list scheduling, a very simple and quickly calculated metric has been used to weigh the different operations that are ready to be scheduled. The weight is simply the pre-calculated ASAP value subtracted by the current control step value. This could yield negative values, but the algorithm is indifferent to this, since it only compares the values. The approach has proven to be very efficient and yields as good schedules as compared to using more advanced metrics, which are harder to compute [18]. The scheduler is used not only to compute the hardware execution time estimates, but also to generate final schedules that could be used in the hardware synthesis.

Looking at figure 12, it is clear that the execution time estimate of a leaf BSB (note: a leaf in the fully expanded BSB hierarchy) is easily determined from the schedule of a ConGIF CDFG. To obtain the final hardware execution time for each leaf BSB, $t_{h,i}$, the schedule length (number of timesteps) of each leaf BSB is multiplied with the profiling count of the BSB.

Execution times for higher level constructs such as loop BSBs and conditional BSBs are obtained by summing the execution times of their child BSBs. Note that we do not compute the execution time of a conditional as the maximum of the execution time of each of its branches. Since we have a profile on the conditional, we know how many times each of the branches are executed and therefore the total execution time for the conditional must be the sum of the execution times of the branches.

Finally, the number of timesteps used in the schedule is used to estimate the hardware controller area, $a_{h,i}$, for each leaf BSB as the number of timesteps is equal to the number of states in the controller [37].

7.3. Hardware Execution Time for the Straight Example

We have used the hardware execution time estimator to schedule the Straight example for three different sets of resource allocations. The results are shown in table 1. The execution time estimate takes the profile into account, but for the Straight example, the profile is very simple as discussed in section 5.3. Note the typical trade-off between execution time and datapath area.

Table 1. Hardware execution time estimates for the Straight example

Adder/subtracter	*Multipliers*	a_{dp}	*Number of cycles*
1	1	472	231
2	2	908	133
1	2	569	135

8. Software Modeling and Estimation

Software execution time for a pure DFG (i.e. no controlflow) is estimated by performing a topological sort (linearization) of the nodes in the DFG. The operations of the nodes are then mapped to a generic instruction set with the addressing modes of the instructions determined by data-dependencies and a greedy register allocation scheme.

Figure 17 shows three equally valid linearizations for the same DFG. Different linearizations result in the same number of control steps but have potentially different register requirements. This may result in different mixtures of addressing modes and intermediate register-to-memory and memory-to-register `move` instructions for

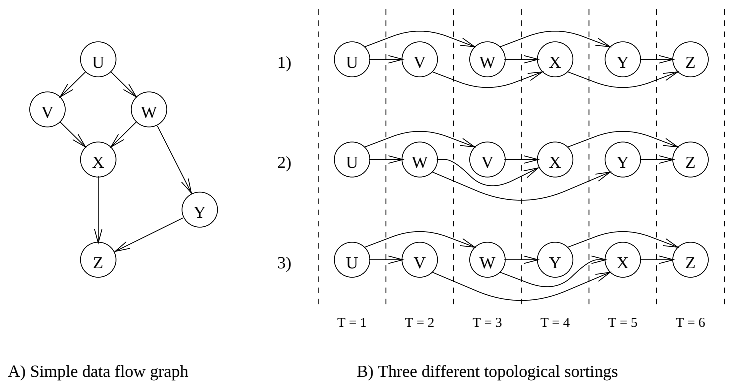
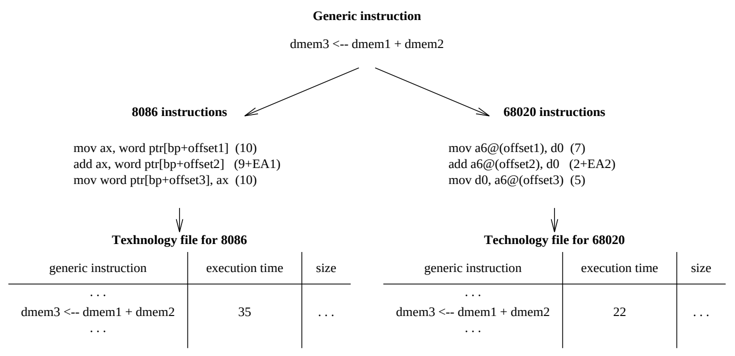

U
V
W
X
Y
Z
1)
2)
3)
T = 1
T = 2
T = 3
T = 4
T = 5
T = 6
A) Simple data flow graph
B) Three different topological sortings

Figure 17. Example of different DFG linearizations. Partly from [20], page 27, fig. 10

the software instructions that correspond to the graph nodes and thus in different execution times and object code sizes. We do not currently model the linearization strategy or register allocation scheme of any particular compiler, but will do so as we develop our own cross compiler. The linearization is performed using the topological sort algorithm from [34].

The execution times of the generic instructions to which the DFG nodes are mapped, are determined from a technology file corresponding to the target microprocessor, as illustrated in figure 18. This is similar to the approach described



```
mov ax, word ptr[bp+offset1]  (10)
add ax, word ptr[bp+offset2]  (9+EA1)
mov word ptr[bp+offset3], ax  (10)
```

```
mov a6@(offset1), d0  (7)
add a6@(offset2), d0  (2+EA2)
mov d0, a6@(offset3)  (5)
```

generic instruction	execution time	size
. . .		
dmem3 <-- dmem1 + dmem2	35	. . .
. . .		

generic instruction	execution time	size
. . .		
dmem3 <-- dmem1 + dmem2	22	. . .
. . .		

Figure 18. Execution-time of a generic instruction for different processors([17], page 6, fig.3)

in [17], where good estimation results are reported, and we use the same technology files for the 8086, 80286, 68000 and 68020 microprocessors. The advantages as compared to an approach where a specific compiler is used for each processor is that it is not necessary to make/find a new compiler for each new processor that is considered and that "compilation" is faster. On the other hand, the eventual optimizations of the compiler actually used in the co-synthesis process are not modeled very accurately.

The execution time of the whole DFG is obtained by summing the execution times of the generic instructions. This sum is multiplied with the profiling count for the DFG. Execution times for higher level constructs such as loop BSBs and branch BSBs are obtained as described in section 7.2.

8.1. Software Execution Time for the Straight Example

We have used the estimation scheme described above to obtain execution time estimates of the Straight example for each of the supported processors. The results are shown in table 2.

Table 2. Software execution time estimates for the Straight example

Micro-processor	*Number of cycles*
8086	8647
80286	1848
68000	5166
68020	2988

9. Communication Estimation

Communication is currently assumed to be memory mapped I/O. The transfer of n values from software to hardware is assumed to require n generic `MOV` microprocessor instructions and n `Import` operations as defined in the hardware library. Communication from hardware to software is estimated in the same way, just using the hardware `Export` operation instead. Estimating the number of values n will be explained in the next section.

10. Hardware/Software Partitioning

This section presents the PACE algorithm [37], [39] which is used to obtain fast functional partitions. The idea behind the algorithm is that it should be able to handle a semi-global view of communication and hardware sharing while still being relatively fast as compared to heuristics which are capable of optimizing for global communication, global compiler optimizations, etc.

10.1. The Partitioning Model

The partitioning model which the PACE algorithm uses is illustrated in figure 19. In this model, hardware BSBs and software BSBs cannot execute in parallel. Furthermore, adjacent hardware BSBs are assumed to be able to communicate the read/write variables they have in common directly between them without involving the software side. As illustrated in the figure, a given hardware/software partition can be thought of as composed of sequences of adjacent BSBs which only communicate their *effective* read- and write-sets from/to the software side. The following definitions formalize these assumptions.

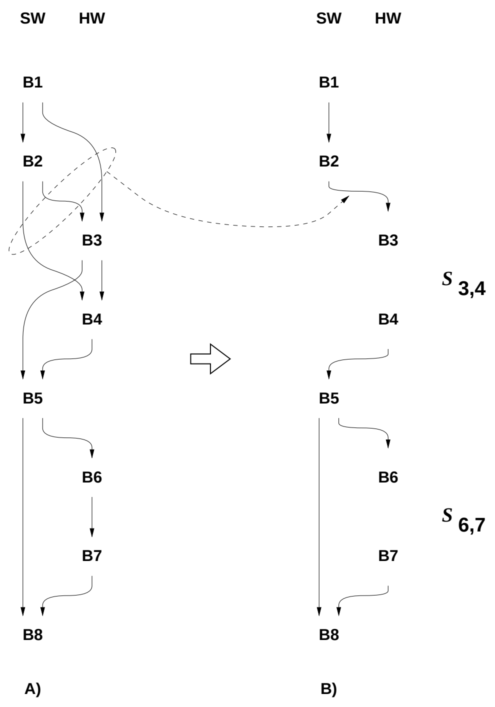


Figure 19. Partitioning model used by PACE: a) Example of actual data-dependencies between hardware- and software BSBs, b) How data-dependencies between adjacent hardware BSBs and software BSBs are interpreted in the model.

Definition. $S_{i,j}$, $j \geq i$, denotes the sequence of BSBs $\langle B_i, B_{i+1}, \ldots, B_j \rangle$.

Definition. The effective read-set and the effective write-set of a sequence $S_{i,j}$ are denoted $r_{i,j}$ and $w_{i,j}$ respectively and are defined as [1]

$$\begin{aligned} r_{i,j} &= (r_i \cup r_{i+1} \cup \cdots \cup r_j) \setminus (w_i \cup w_{i+1} \cup \cdots \cup w_j) \\ w_{i,j} &= (w_i \cup w_{i+1} \cup \cdots \cup w_j) \setminus (r_i \cup r_{i+1} \cup \cdots \cup r_j) \end{aligned}$$

Using these definitions and the BSB definitions given in section 5.2 we can now compute the speedup induced by moving a sequence of BSBs from hardware to software. Note, that in this paper we will use the term "speedup" to denote the *absolute* time saved by moving functionality to hardware and not the relative speedup which is otherwise commonly used.

Definition. The total (possibly negative) speedup induced by moving a BSB sequence $S_{i,j}$ to hardware is denoted $s_{i,j}$ and is computed as

$$s_{i,j} \;=\; \sum_{k=i}^{j} \mathrm{pc}(B_k)(t_{s,k} - t_{h,k}) - (\sum_{v \in r_{i,j}} \mathrm{ac}(v) t_{s \to h} + \sum_{v \in w_{i,j}} \mathrm{ac}(v) t_{h \to s})$$

where $t_{s \to h}$ and $t_{h \to s}$ denote the software-to-hardware and hardware-to-software communication times for a single variable, respectively.

Definition. The area penalty $a_{i,j}$ of moving $S_{i,j}$ to hardware is computed as the sum of the individual BSB areas (note, the area of the BSB is the area used to implement the BSB controller, c.f. section 7.1) is computed as :

$$a_{i,j} \;=\; \sum_{k=i}^{j} a_{h,k}$$

In section 7.1 we discussed how the effect of hardware sharing is taken into account. Note that in calculating the speedup and area of a sequence it is not considered that hardware synthesis may synthesize the sequence as a whole which would probably reduce both sequence area and execution time as compared to just summing the individual area- and execution time components as described above. Incorporating such sequence optimizations in the estimations will be fairly straightforward but has not been carried out yet. Note, however, that the improvement in speedup induced by all BSBs within the sequence being able to communicate directly with each other *is* taken into account.

10.2. The Partitioning Problem

The partitioning problem can now be formulated as that of finding the combination of non-overlapping hardware sequences which yields the best speedup while having a total area penalty less than or equal to the available hardware controller area $\mathcal{A}$.

The problem is best illustrated by an example. Figure 20 shows four BSBs which must be partitioned as to reach the largest speedup on the available area $\mathcal{A}$=3. The speedup and area penalty for a single BSB which is moved to hardware without considering interactions with neighboring BSBs is shown below each BSB. The numbers between two BSBs denote the extra speedup which is incurred because of the BSBs being able to communicate directly with each other when they are both placed in hardware. If A for example is placed in hardware, then placing B

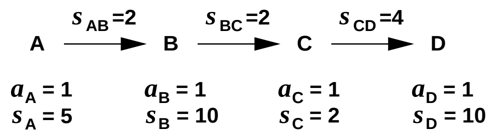


Figure 20. Example of partitioning problem with communication cost considerations.

in hardware as well will increase the speedup by 2 due to the fact that A and B no longer have to communicate across the hardware/software boundary.

Obviously B and D should be placed in hardware as they have large inherent speedups, each equal to 10. This leaves room for one more BSB. Should it be A or C? The answer to this is not obvious as A induces a large inherent speedup (5) but a small communication speedup (2) when placed together with B in hardware, whereas C induces a smaller inherent speedup (2) but on the other hand induces a large communication speedup $(2 + 4)$ when placed together with B and D in hardware. The following section explain how the PACE algorithm solves this problem.

10.3. The PACE Algorithm

The algorithm utilizes the previously mentioned fact, that any possible partition can be thought of as composed of *sequences* of BSBs. If A, C and D are chosen for hardware, it corresponds to choosing the sequences $S_{A,A}$ and $S_{C,D}$. The speedup of sequence $S_{C,D}$ is larger than the sum of speedups of its components C and D due to the extra communication speedup induced by both blocks being chosen for hardware. So a natural approach will be to calculate the areas and speedups of *all sequences* of BSBs, and chose the combination of sequences that induces the largest speedup. The areas and speedups of all sequences are calculated and shown in table 3. The ordering and grouping of BSBs is explained below.

The problem is to find the combination of non-overlapping sequences which fit within the available area $\mathcal{A}$ and whose speedup sum is as large as possible. This problem cannot be solved with an ordinary Knapsack Stuffing algorithm as some of the sequences are mutually exclusive (because they contain identical BSBs) and therefore cannot be moved to hardware at the same time. But if the sequences are ordered and grouped as shown in the table, a dynamic programming algorithm *can* be constructed which does not attempt to chose mutually exclusive sequences for hardware at the same time.

Before describing how the algorithm works, it is necessary to define the list of trial areas:

Definition. Let δ be an integer value called the area granularity, then the list of trial areas is defined as:

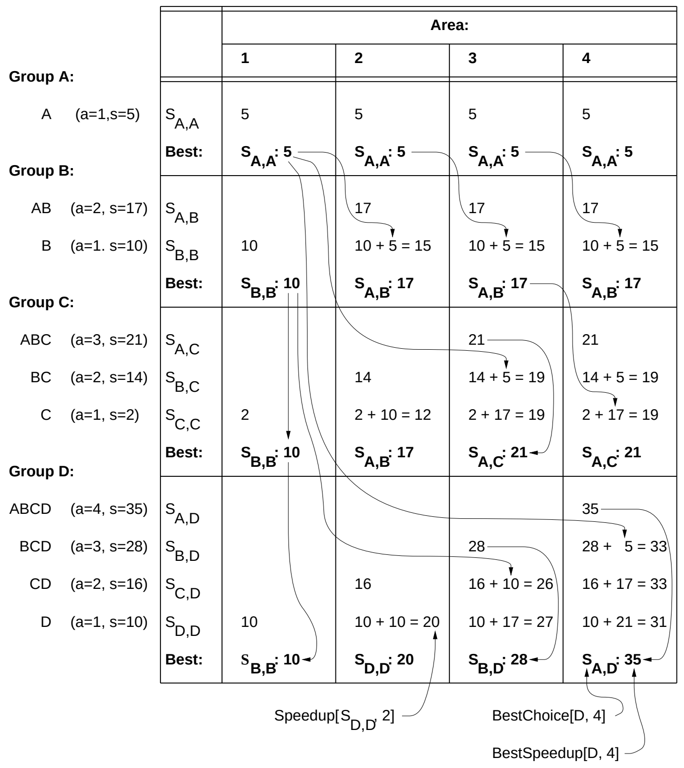
$$\mathcal{A}_T \;=\; < a_0, a_1, \ldots, a_i, \ldots, a_m >$$

Table 3. Grouping of sequences.

Sequence	**Elements**	**Area**	**Speedup**
Group A: All sequences ending with A			
$S_{A,A}$	A	1	5
Group B: All sequences ending with B			
$S_{A,B}$	AB	2	17
$S_{B,B}$	B	1	10
Group C: All sequences ending with C			
$S_{A,C}$	ABC	3	21
$S_{B,C}$	BC	2	14
$S_{C,C}$	C	1	2
Group D: All sequences ending with D			
$S_{A,D}$	ABCD	4	35
$S_{B,D}$	BCD	3	28
$S_{C,D}$	CD	2	16
$S_{D,D}$	D	1	10

where $a_i = i \cdot \delta$ and $a_m = \mathcal{A}$.

If for exampel $\mathcal{A} = 10$ and $\delta = 2$ then the list of trial areas is $< 0, 2, 4, 6, 8, 10 >$. Note, that the individual BSB areas must be multiples of δ. The algorithm works as follows. Assume first that for each group up to and including group C the best (maximum speedup) combination of sequences has been found and stored for each trial area in $\mathcal{A}_T$. Now consider group D as well. Assume then that for instance sequence $S_{C,D}$ with area $a_{C,D}$ is selected for hardware at the available area a. How is the optimal combination of sequences on the remaining area $a - a_{C,D}$ then found? As C and D have been chosen for hardware, only A and B remain. So the best solution on the remaining area must be found in group B which contains the best combination of sequences for all BSBs from the set {A,B}. Similarly, if the "sequence" $S_{D,D}$ is chosen for hardware, the best combination on the remaining area is found in group C. The optimal combination is always found in the group whose letter in the alphabet comes immediately before the letter of the first index in the chosen sequence. The important thing to note is that when a sequence from group X has been chosen, the optimal combination of sequences on the remaining area can be found in one of the groups A to *pred*(X), and, when sequences are selected as above, no mutually exclusive BSBs are selected simultaneously. In this way the best solutions for a given group can always be determined on basis of the best solutions found for the previous groups.



			Area:			
			1	2	3	4
Group A:						
A	(a=1,s=5)	$S_{A,A}$	5	5	5	5
		Best:	**$S_{A,A}$: 5**	**$S_{A,A}$: 5**	**$S_{A,A}$: 5**	**$S_{A,A}$: 5**
Group B:						
AB	(a=2, s=17)	$S_{A,B}$		17	17	17
B	(a=1. s=10)	$S_{B,B}$	10	10 + 5 = 15	10 + 5 = 15	10 + 5 = 15
		Best:	**$S_{B,B}$: 10**	**$S_{A,B}$: 17**	**$S_{A,B}$: 17**	**$S_{A,B}$: 17**
Group C:						
ABC	(a=3, s=21)	$S_{A,C}$			21	21
BC	(a=2, s=14)	$S_{B,C}$		14	14 + 5 = 19	14 + 5 = 19
C	(a=1, s=2)	$S_{C,C}$	2	2 + 10 = 12	2 + 17 = 19	2 + 17 = 19
		Best:	**$S_{B,B}$: 10**	**$S_{A,B}$: 17**	**$S_{A,C}$: 21**	**$S_{A,C}$: 21**
Group D:						
ABCD	(a=4, s=35)	$S_{A,D}$				35
BCD	(a=3, s=28)	$S_{B,D}$			28	28 + 5 = 33
CD	(a=2, s=16)	$S_{C,D}$		16	16 + 10 = 26	16 + 17 = 33
D	(a=1, s=10)	$S_{D,D}$	10	10 + 10 = 20	10 + 17 = 27	10 + 21 = 31
		Best:	**$S_{B,B}$: 10**	**$S_{D,D}$: 20**	**$S_{B,D}$: 28**	**$S_{A,D}$: 35**

Figure 21. The PACE algorithm employed for a simple example.

Figure 21 shows how the best combination of sequences can be found using three matrices; `Speedup[1..`n_S`, 0..`$\mathcal{A}$`]`, `BestSpeedup[1..`n`,1..`$\mathcal{A}$`]` and `BestChoice[1..`n`, 0..`$\mathcal{A}$`]`.

n_S is the number of sequences, n is the number of BSBs and $\mathcal{A}$ is the available area. Zero entries are not shown. Arrows indicate where values are copied from, but arrows are not shown for all entries in order to make the figure more readable.

The `Speedup` matrix contains for each sequence and each available area the best speedup that can be achieved if that sequence is first moved to hardware and then sequences from the *previous* groups are moved to hardware. In the figure, `Speedup[S`$_{B,C}$`, 3]` is 19 and is found as the inherent speedup of $\mathrm{S}_{B,C}$ which is 14 plus the best obtainable speedup 5 on area $3 - a_{B,C} = 3 - 2 = 1$ in group A (as B and C have been chosen).

The `BestSpeedup` matrix contains for each group g (which there are n of) and each area the best speedup that can be achieved by first selecting a sequence from that group or one of the previous groups. It can be calculated as

$$\texttt{BestSpeedup[}g\texttt{, }a\texttt{]} = \max\left(\max_{S\in g}(\texttt{Speedup[}S\texttt{,}a\texttt{]}), \texttt{BestSpeedup[}pred(g)\texttt{,}a\texttt{]}\right)$$

The `BestChoice[`g`,`a`]` matrix identifies the choice of sequence that gave this maximum value. The last two matrices are interleaved and typeset with bold letters in the figure.

In the example, `BestChoice[C, 3]` is 21 as this is the maximum speedup that can be found in group C with available area 3 and it is larger than the largest speedup that could be found in the previous groups, namely 17. The corresponding choice of sequence is $S_{A,C}$. In contrast, `BestSpeedup[C,1]` and `BestChoice[D,1]` are copied from the corresponding entries of the previous group. For group C this is because all `Speedup` entries in that group for area 1 are smaller than the best speedup 10 achieved with only sequences from groups up to and including B. For group D, `Speedup[S`$_{D,D}$`, 1]` is also 10, so the choice of best sequence for this group is arbitrary.

The solution to the posed problem from section 10.2 is found in the `BestChoice[D, 3]` and `BestSpeedup[D,3]` entries. The best initial choice is sequence $S_{B,D}$ with the corresponding total speedup of 28. As the area of this sequence is 3, no other sequences were taken, and need thus not be found by backtracking. This shows that it was best to chose C for hardware instead of A. The area 4 was included in the figure to show that the algorithm correctly chooses all four BSBs for hardware when there is room for them. This can be seen from the [D,4] entries.

Once the `BestSpeedup` and `BestChoice` lines have been calculated for each group, the `Speedup` values are no longer needed. Actually, the `Speedup` matrix is not needed at all, as it can be replaced by the `BestSpeedup` matrix whose maximum values can be calculated "on the run". This is because we are only interested in maximum values and corresponding choice of sequences for each group. This means that instead of the memory requirements being proportional to the number of sequences n_S, they are now proportional to n, as only the `BestSpeedup` and `BestChoice` matrices are needed. The PACE algorithm is shown in figure 22.

After the algorithm has been run, the best speedup that can be obtained is found in the entry `BestSpeedup[NumBSBs, AvailableArea]`. But as for the simple Knapsack algorithm, reconstruction of the chosen sequences and thereby of the chosen BSBs is necessary.

10.4. Algorithm Analysis

Direct inspection of the PACE algorithm shows that the time complexity is $O(n^2 \cdot len(\mathcal{A}_T))^2$ and the space complexity is $O(n \cdot len(\mathcal{A}_T))$ (the PACE-reconstruct algorithm obviously has smaller time and area complexity and can hence be disregarded). Note that areas and speedups must be expressed as integral values. $len(\mathcal{A}_T)$ can be reduced (at the expense of partitioning quality) by using a larger

```
PACE ($n, \mathcal{A}$) ≡
{
 /* Initialization */
 for_all groups g = 1 to n do
  for_all areas a = 0 to $\mathcal{A}$ do {
   BestChoice[g,a] ← {};
   BestSpeedup[g,a] ← 0;
  }
 /* Partitioning */
 for_all groups g = 1 to n do {
  /* Find the best solutions for all groups up to g */
  /* All seqs. in group g have high index g: */
  HighBSB ← g;
  for LowBSB = 1 to HighBSB do {
   /* Traverse the BSB seqs. in group g */
   SeqArea ← total area of seq. $S_{LowBSB,HighBSB}$;
   SeqSpeedup ←
    total speedup of seq. $S_{LowBSB,HighBSB}$;
   for_all areas a = SeqArea to $\mathcal{A}$ do {
    /* A is such that there is room for the seq. */
    /* Assume that the sequence is selected for */
    /* hardware. If LowBSB = 1, the sequence */
    /* consists of all elements, and no more */
    /* elements can be selected for hardware. */
    if (LowBSB = 1) then {
     /* Only this sequence can be selected */
     if SeqSpeedup > BestSpeedup[g, a] then {
      BestSpeedup[g,a] ← SeqSpeedup;
      BestChoice[g,a] ← $S_{LowBSB,HighBSB}$;
     }
    }
    else {
     /* Also assume that the best solution for */
     /* sequences up to LowBSB-1 */
     /* on the remaining area is selected. */
     if (SeqSpeedup +
        BestSpeedup[LowBSB-1, a-SeqArea]) >
          BestSpeedup[g, a] then {
      BestSpeedup[g,a] ← SeqSpeedup +
          BestSpeedup[LowBSB-1, A-SeqArea];
      BestChoice[g,a] ← $S_{LowBSB,HighBSB}$;
     }
    }
   }
  }
  /* For each area, now see if the best solution */
  /* found for sequences without the element */
  /* HighBSB is better than the solution just */
  /* found for sequences with HighBSB. If so, */
  /* replace the just found solutions with the */
  /* previously found better solutions. */
  if (HighBSB > 1)
   for_all areas a = 0 to $\mathcal{A}$ do
    if BestSpeedup[g-1, a] > BestSpeedup[g, a] then {
     BestSpeedup[g, a] ← BestSpeedup[g-1, a];
     BestChoice[g, a] ← BestChoice[g-1, a];
    }
 } /* for_all groups */
 return BestChoice[], BestSpeedup[];
} /* Algorithm */
```

```
PACE-reconstruct ($n$, $\mathcal{A}$,
        BestSpeedup[], BestChoice[]) ≡
{
 /* Initialization */
 HwBSBList ← {};
 /* The best execution-time on the */
 /* given area has been found. The */
 /* same execution-time may be */
 /* achievable for a smaller area. */
 /* Reconstruct the sequences that */
 /* lead to best execution-time, */
 /* occupying the smallest area. */
 AStart ← 0;
 Found ← false;
 while (AStart <= $\mathcal{A}$) and
         not Found do {
  if BestSpeedup[$n$, AStart] =
     BestSpeedup[$n, \mathcal{A}$] then
   Found ← true
  else
   AStart ← AStart + 1;
  {
 /* The best starting entry has been */
 /* found. Now reconstruct the */
 /* sequences. */
 a ← Astart;
 g ← $n$;
 repeat {
  Seq ← BestChoice[g, a];
  if Seq <> {} then {
   LowBSB ← first index of Seq;
   HighBSB ← second index of Seq;
   for BSB = LowBSB to HighBSB do
    add BSB to HwBSBList;
   a ← a − area(Seq);
   g ← LowBSB − 1;
  }
 } until (a < 0) or (Seq = {}) or
        (g = 0);
 return HwBSBList;
}
/* Algorithm */
```

Figure 22. PACE and reconstruct algorithm

area granularity, δ. The n^2 term can be reduced by enlarging BSB granularity by hierarchical collapsing or by only considering BSB sequences which induce a speedup greater than zero (or greater than some given percentage). Also, there is no need to precalculate areas and speedups for sequences whose total area will be greater than $\mathcal{A}$.

As for the simple Knapsack problem, the dual problem of minimizing area with a fixed time-constraint can be solved by swapping the area- and speedup entries calculated for each sequence. Another approach could be to scan the bottom line of the `BestSpeedup` matrix from the left (see figure 21) until a entry is found which violates the time-constraint, as the PACE algorithm in effect calculates the best speedup for *all* areas.

Note that the areas and speedups of all sequences must be precalculated before the algorithm can execute. This operation has time complexity $O(n^2)$.

10.5. Partitioning the Straight Example

The example application has been partitioned with the partitioning tool in our tool suite. The hardware allocation is one adder/subtracter and one multiplier. The total hardware area is 1377 FPGA modules and the estimated hardware datapath area is 811 FPGA modules which leaves 566 modules for BSB controllers. The micro-processor is an MC68000.

Table 4. Partitioning results for the Straight example.

Partition algorithm	*System execution time*	*Improvement*	*Hardware controller area*
PACE, adj. block comm.	302	1611%	510

11. Design Space Exploration with LYCOS

As explained in the introduction, the design space exploration phase requires repeated use of a partitioning algorithm because the best functional partition must be found for every considered target architecture. Having found the best partitioning for each target architecture in terms of execution time, hardware area, etc., the individual target architectures can be compared taking other criterias into account, such as price, power consumption, etc., in order to choose the best suited target architecture.

The design space can be very large (software processor, hardware chip, communication protocol, hardware configuration, etc., must be chosen). The partitioning algorithm used in this phase must therefore be a fast one which unfortunately means that it does not necessarily result in a functional partition which is as good as the one a more accurate, but slower partitioning algorithm can produce. This is because fast partitioning algorithms typically use a simplified partitioning model where for example functions are flattened, hardware/software communication overhead is ignored or only estimated locally on BSB basis, hardware area calculated locally on BSB basis disregarding hardware compiler optimizations over BSB boundaries, etc.

When the design space exploration phase has come up with one or several good target architectures and corresponding partitions by using a fast partitioning algorithm, a more advanced (and slower) partitioning algorithm can now be used (but now only a single or few times) on this or these target architectures. This algorithm may be a heuristic which can handle global aspects that the less accurate but fast partitioning algorithm cannot. Examples are optimization for global communication using local hardware store, estimation of hardware interconnect (including MUXes, buses, etc.) and register area (which can only be done for whole hardware partitions), estimation of global software compiler optimizations, analysis of function calls, etc.

The more advanced algorithm results in a partition which, when evaluated in accordance with the more advanced partitioning model it employs, will probably turn out to be better than the partition the more simple partitioning algorithm came up with (when evaluated in accordance with the same, more advanced partitioning model). The best of the two partitions should now be chosen as basis for the next phase which is heterogeneous code generation, i.e. generation of object code and communication code for the software part of the partition and generation of communication and calculation hardware for the hardware partition.

The partitioning and design space exploration tool [37] in the LYCOS system provides for the design space exploration scheme described above by including simple and fast partitioning algorithms such as the Knapsack Stuffing algorithm [12][3] and the more advanced PACE algorithm. Heuristics which allow for more global optimizations have not yet been implemented.

11.1. Comparing Different Partitioning Algorithms

This section describes various aspects of using partitioning algorithms which employ too simplified communication models. This is done by comparing the simple Knapsack algorithm with the PACE algorithm.

The sample application used for these comparisons is a VHDL behavioral description, taken from an image processing application in optical flow [40]. The application takes a sequence of satellite images of cloud movements used for weather forecast and generates 10 new images in between each original image in order to have a sequence with smooth cloud movements. The image processing consists of three stages; extraction of motion invariants, local measurement of visual motion, and integration of local measurements in conjunction with a priori knowledge. The application used in this paper is taken from the second stage which calculates eigenvectors in order to obtain local orientation estimates for the cloud movements. It consists of 448 lines of behavioral VHDL. The corresponding CDFG contains 1511 nodes and 1520 edges. BSB software execution-time is estimated for a 8086 processor and a hardware library for an Actel ACT 3 FPGA is used to estimate hardware datapath and controller area. Partitioning is performed for a sequence of total hardware areas ranging from 1000 to 2000 in steps of 20, where an area unit equals the area of a logic/sequential module in the FPGA. Table 5 summarizes the characteristics of the most important modules.

Table 5. Area and execution-time estimates for hardware modules and operations.

Module	*Operations*	*Area*	*Cycles*
add-sub-comb	add, subtract, less, equal	97	1
mul-comb	multiply	339	4
mul-ser	multiply	103	16
div-comb	divide	339	4
div-ser	divide	103	16

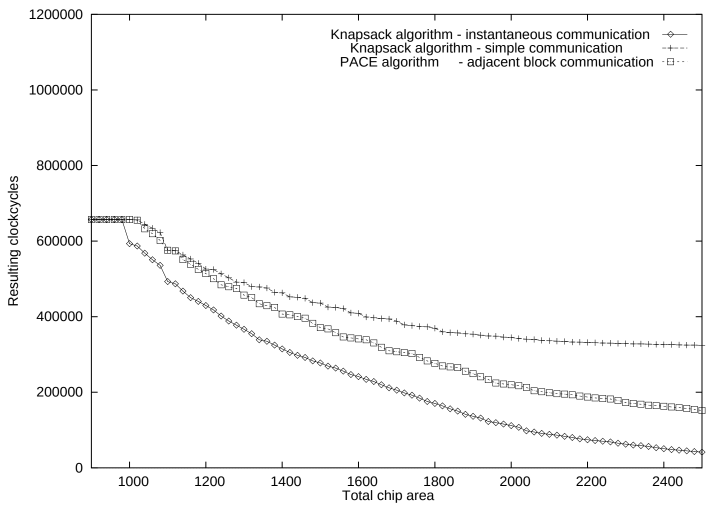


Figure 23. The PACE algorithm compared with Knapsack algorithms when the partitioning results are evaluated according to the samme model as used for the partitioning.

Figure 23 shows the results of partitioning the sample application using three different partitioning models (see figure 24); instantaneous communication, simple communication where the read- and write-sets of a BSB are *always* transferred regardless of other BSBs placed in hardware, and adjacent block communication which is the one used by the PACE algorithm.

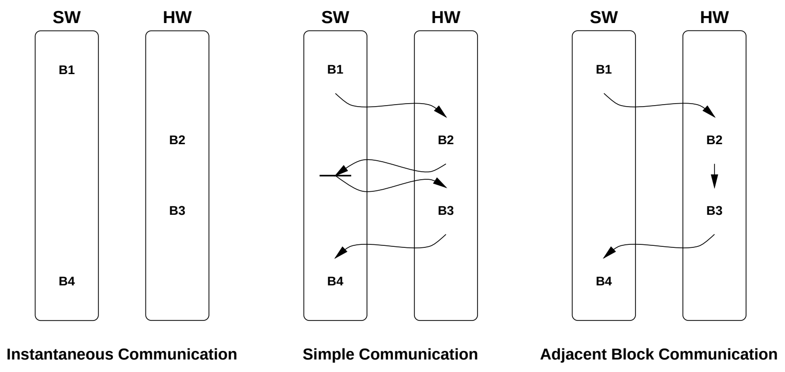


Figure 24. The three partitioning models used in the partitioning algorithm comparison.

For chip areas less than the allocated area for the datapath (760 in figure 23 as allocation A from table 6 is used), no speedup is obtained as no controller area is available. As soon as controller area is available, the approach which assumes instantaneous communication starts to move BSBs to hardware. For the approaches that take communication into account, moving BSBs to hardware is not beneficial before the total area reaches about 1040. It can be seen that as the chip area increases, more and more BSBs are moved to hardware, thus, for large chip areas there is less communication between hardware and software, hence PACE and the approach assuming instantaneous communication become comparable. From the figure it is clear that the approach using the simple communication model does not move as many BSBs to hardware as the PACE which takes adjacent block communication into account. This is mainly due to the fact that many of the BSBs have a communication overhead which is larger than the speedup they induce.

Just comparing the curves of figure 23 may lead to the conclusion that assuming instantaneous communication is the best approach. This is due to the fact that the algorithm assumes that the partition will be implemented using instantaneous communication. However, as this is unrealistic, the partitioning results will have to be evaluated according to the most realistic implementation, i.e. using the adjacent block communication model which assumes local hardware store. This results in the curves shown in figure 25.

The first thing that is noted is, that even though the approach based on instantaneous communication in figure 23 "thinks" it is achieving a large speedup for areas above 1000, it is actually producing worse than the all-software solution (as seen in figure 25).

The experiments in this section have shown that the best results are (not surprisingly) obtained by partitioning according to a model which resembles characteristics of the target architecture, i.e. the adjacent block communication model, and that the partitions which are produced by a partitioning algorithm which uses a sim-

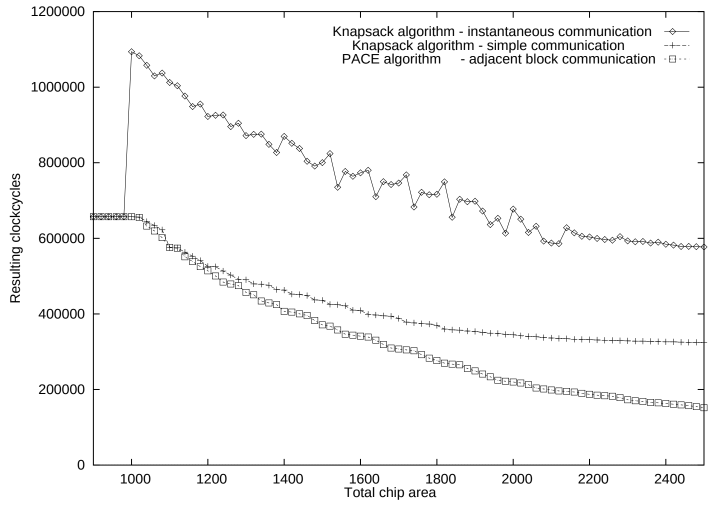


Figure 25. The PACE algorithm compared with Knapsack algorithms which assume instantaneous communication or do not account for adjacent hardware block communication optimization. The results are obtained using allocation A from table 6.

ple partitioning model should always be evaluated in accordance with a *realistic* implementation model [38].

11.2. Design Space Exploration of Two Examples

This section describes experiments with the optical flow example and the Straight example which illustrate how design space exploration is performed in LYCOS. The experiments concentrate on the problem of selecting a good hardware configuration. Of course the LYCOS system can also be used to experiment with different software processors.

Figure 26 shows the results of partitioning the optical flow application with the PACE algorithm for three different allocations; A, B and C, all listed in table 6.

Widely different results are obtained for given available areas, and a specific allocation which is optimal for all areas cannot be found. Allocation C has the smallest datapath area which means that for relatively small areas, the partitioning algorithm is able to move BSBs to hardware and, thus, obtain the best partition. Around the area 1500 this is changed, now allocation A becomes more attractive due to the fact that the larger datapath allocation can benefit from the inherent parallelism of the sample application, i.e. larger speedups may be achieved for the individual BSBs. The figure also illustrates the problem of allocating too much datapath area, leaving little area for BSB controllers, as allocation B which has the

Table 6. Modules and corresponding area for each of the three allocations.

	Allocation		
Module	*A*	*B*	*C*
add-sub-comb	2	1	1
mul-comb	1	0	0
mul-ser	0	8	1
div-comb	1	0	0
div-ser	0	1	1
Area	760	1148	427

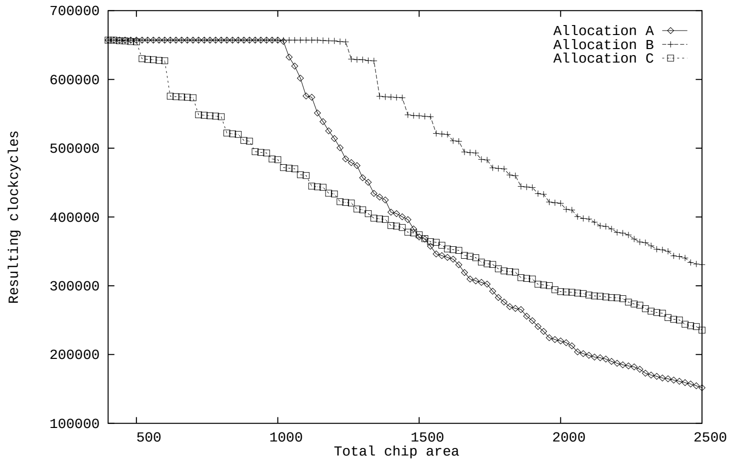


Figure 26. The PACE algorithm employed for different allocations.

largest datapath area never manages to give the best partitioning even for large chip areas.

Figure 27 illustrates the controller/datapath area tradeoff when partitioning the Straight example under hardware area constraints. The figure shows the all hardware execution times for three different allocations, containing 1, 2 and 3 combinatorial multipliers respectively, and the corresponding obtained speedup. Going from 1 to 2 multipliers results in a significant speedup which would be expected as the all-hardware execution time is almost halved. However, the expected speedup when adding an extra multiplier (reducing the all-hardware execution time further)

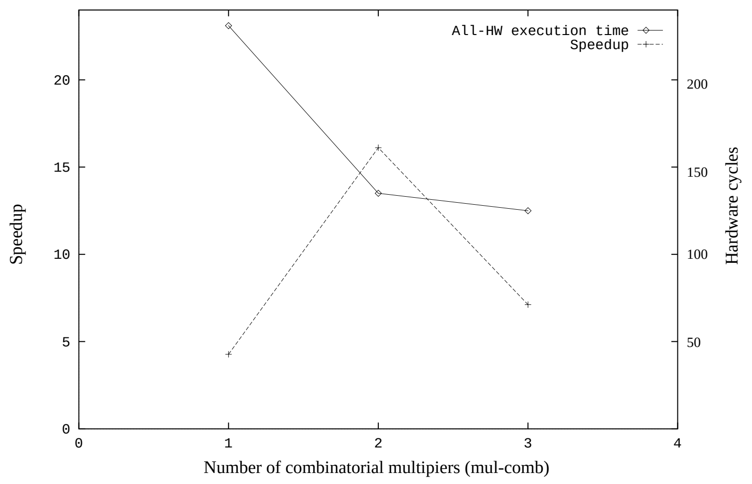


Figure 27. Controller/datapath area trade-offs when partitioning under a hardware area contraint.

is *not* seen. Instead the speedup is reduced to around the same level as for a single multiplier. The reason for this is of course that 3 multipliers allocate a large fraction of the available hardware area, leaving only a small area for the controller, and hence only a few BSBs can be moved to hardware.

12. Summary and Future Work

In this paper we have presented the LYCOS system which is an experimental cosynthesis environment.

We have shown how LYCOS can be used for hardware/software partitioning of an application onto a target architecture consisting of a single CPU and a single hardware component. Given a C or VHDL specification of the application (a function/process), a set of typical inputs for this, a specification of the microprocessor and a specification of the hardware library and allocation (number of available hardware modules from the given hardware library), the partitioning tool comes up with a partition which, according to a certain partition model, is found optimal with respect to the execution time. LYCOS provides the choice between different partition models and partitioning algorithms, one of which is a novel algorithm, called PACE. It is characterized by recognizing that adjacent blocks placed in hardware may share hardware modules as they execute in mutual exclusion and may communicate variables directly between them without involving the software side.

Finally, we illustrated how LYCOS can be used for design space exploration: For a given application, in order to find the best target architecture (with respect to parameters like execution time, hardware area and price) the partitioning is performed for many different target architectures (i.e. different specifications of microprocessors, hardware libraries and allocations).

Topics for ongoing and future work are:

- the support of additional input specification languages like RSL and Synchronized Transitions
- extension of the internal CDFG format such that more input language constructs can be translated
- multiple communicating processes
- more general target architectures than single CPU, single hardware processor architectures
- code generation

The on-going and future activities and developments on the LYCOS system can be watched at WWW, http://www.it.dtu.dk/~lycos.

Acknowledgments

This research has been sponsored by the Danish Technical Research Council under the "Codesign" program.

Notes

1. "\" is the normal set difference operator.
2. *len*(X) returns the number of elements in the list X.
3. The Knapsack Stuffing algorithm can be used without modification in the LYCOS system as only leaf BSBs which can always be moved to/from hardware independently of each other are considered (see figure 13 on page 17).

References

1. S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto. Hw/sw codesign for embedded telecom systems. In *proceedings of ICCD'94*, October 1994.
2. S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto. The role of VHDL within the TOSCA hardware/software codesign framework. In *EURO-DAC '94*, pages 612–617, 1994.
3. E. Barros and W. Rosenstiel. A clustering approach to support hardware/software partitioning. In J. Rozenblit and K. Buchenrieder, editors, *Codesign*, pages 230–262. IEEE Press, 1995.
4. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79(9):1270–1282, September 1991.
5. P. Bojsen. *Formalizing data flow graphs.* PhD thesis, Department of Computer Science, the Technical University of Denmark, 1994.

6. J.P. Brage. *Foundations of a High-Level Synthesis System.* PhD thesis, Department of Computer Science, the Technical University of Denmark, 1994.
7. J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, 4:155–182, 1994.
8. R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1-2):5–50, January 1996.
9. K.M. Chandy and J. Misra. *A Foundation of Parallel Programs Design.* Printice Hall, 1988.
10. M. Chido, D. Engels, H. Hsieh P. Giusto, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided co-design of embedded controllers. *Design Automation for Embedded Systems*, 1(1-2):51–67, January 1996.
11. P.H. Chou, R.B. Ortega, and G. Borriello. The Chinook hardware/software co-synthesis system. In *the 8th International Symposium on System Synthesis*, September 1995.
12. T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1992.
13. J. T. J. Eijndhoven, C. G. de Jong, and L. Stok. The ASCIS data flow graph: semantics and textual format. Technical Report 91-E-251, Eindhoven University of Technology, June 1991.
14. R. Ernst, J. Henkel, and T. Benner. Hardware-software codesign of embedded controllers based on hardware-extraction. In *International Workshop on Hardware-Software Codesign, Estes Park, Colorado*, 1992.
15. R. Ernst, J. Henkel, and T. Benner. Hardware/software co-synthesis of microcontrollers. *Design and Test of Computers*, pages 64–75, December 1992.
16. D.D. Gajski, F. Vahid, and S. Narayan. A system-design methodology: Executable-specification refinement. In *the European Conference on Design Automation*, February 1994.
17. Jie Gong, Daniel D. Gajski, and Sanjiv Narayan. Software estimation from executable specifications. Technical Report ICS-93-5, Dept. of Information and Computer Science, University of California, Irvine, Irvine, CA 92717-3425, March 8 1993.
18. Jesper Grode. Scheduling of control flow dominated data-flow graphs. Master's thesis, Technical University of Denmark, 1995.
19. Jesper Grode and Jan Madsen. Resource considerations during parallel scheduling of large control-flow dominated applications. In *4th International Conference on VLSI and CAD*, 1994.
20. Rajesh K. Gupta and Giovanni De Micheli. System synthesis via hardware-software co-design. Technical Report CSL-TR-92-548, Computer Systems Laboratory, Stanford University, October 1992.
21. R.K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* Kluwer Academic Publisher, 1995.
22. R.K. Gupta, C. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *the 29th Design Automation Conference*, June 1992.
23. Bjarne G. Hald and Jan Madsen. A flexible architecture representation for High Level Synthesis. In *Second Asian Pacific Conference on Hardware Description Languages*, 1994.
24. D. Harel. Statecharts: A visual formalism for complex systems. In *Science of Computer Programming 8.* North-Holland, 1987.
25. A. E. Haxthausen. Translation of C Programs into Quenya CDFGs. Technical report, Department of Computer Science, the Technical University of Denmark, 1995.
26. A. E. Haxthausen. Developing a Translator from C Programs to Data Flow Graphs Using RAISE. In *Proceedings of COMPASS'96.* IEEE Computer Society Press, 1996.
27. D. Herrmann, J. Henkel, and R. Ernst. An approach to the adaptation of estimated cost parameters in the COSYMA system. In *CODES '94*, 1994.
28. T. Ismail, K. O'Brien, and A. Jerraya. Interactive system-level partitioning with PARTIF. In *European Conference on Design Automation*, February 1994.
29. T.B. Ismail, M. Abid, and A. Jerraya. COSMOS: A codesign approach for communicating systems. In *Third International Workshop on Hardware/Software Codesign*, pages 17–24. IEEE Computer Society Press, 1994.

30. Axel Jantsch, Peeter Ellervee, Johnny Öberg, Ahmed Hermani, and Hannu Tenhunen. A case study on hardware/software partitioning. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.
31. Axel Jantsch, Peeter Ellervee, Johnny Öberg, Ahmed Hermani, and Hannu Tenhunen. Hardware/software partitioning and minimizing memory interface traffic. In *EURO-DAC '94*, 1994.
32. Axel Jantsch, Peeter Ellervee, Johnny Öberg, Ahmed Hermani, and Hannu Tenhunen. A software oriented approach to hardware/software codesign. In *Proceedings of the Poster Session of the International Conference on Compiler Construction*, April 1994.
33. K. Jensen. Coloured petri nets. In W. Brauer and W. Reisig, editors, *Petri Nets: Central Models and Their Properties*, pages 248–299. Springer-Verlag, September 1987.
34. A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5:558–562, 1962.
35. Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Third International Workshop on Hardware/Software Codesign*, pages 42–48, September 1994.
36. K.M. Kavi, B.P. Buckles, and U.N. Bhat. Isomorphisms between petri nets and dataflow graphs. *IEEE Transaction on Software Engineering*, SE-13(10):1127–1134, October 1987.
37. Peter V. Knudsen. Fine-grain partitioning in codesign. Master's thesis, Technical University of Denmark, 1995.
38. Peter V. Knudsen and Jan Madsen. Aspects of System Modelling in Hardware/Software Partitioning. In *Seventh IEEE International Workshop on Rapid Systems Prototyping*, pages 18–23, 1996.
39. Peter V. Knudsen and Jan Madsen. PACE: A dynamic programming algorithm for hardware/software partitioning. In *4th International Workshop on Hardware/Software Codesign, Codes/CASHE'96*, March 1996.
40. R. Larsen. *Estimation of Visual Motion in Image Sequences.* PhD thesis, Institute of Mathematical Modelling, the Technical University of Denmark, 1994.
41. Z. Liu, M.R. Hansen, J. Madsen, and J.P. Brage. Real-Time Semantics for Data Flow Graphs. Technical report, Department of Computer Science, the Technical University of Denmark, 1995.
42. Jan Madsen and Bjarne G. Hald. A approach to interface synthesis. In *8. International Symposium on System Synthesis*, 1995.
43. Giovanni De Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, 14(4):10–16, August 1994.
44. P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC. *IEEE Trans. Computer-Aided Design*, 8:661–679, 1989.
45. The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice Hall Int., 1992.
46. J. Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.
47. F. Vahid. SLIF: A specification-level intermediate format for system design. Technical Report CS-94-06, Dept. of Computer Science, Univ. of California, Riverside, Sep 1994.
48. F. Vahid and D.D. Gajski. Clustering for improved system-level functional partitioning. In *the 8th International Symposium on System Synthesis*, September 1995.
49. T.Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *International Conference on Computer Aided Design.* IEEE Computer Society Press, 1995.
50. T.Y. Yen and W. Wolf. Sensitivity-driven co-synthesis of distributed embedded systems. In *the 8th International Symposium on System Synthesis*, pages 4–9, September 1995.