VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment

Petru Eles*, Zebo Peng†, and Alexa Doboli*

* Computer Science and Engineering Department
Technical University of Timisoara
Romania

† Dept. of Computer and Information Science
Linköping University
Sweden

Abstract

This paper deals with the problems of system-level specification and partitioning in hardware/software co-design. It first discusses the implication of using VHDL as an implementation-independent specification language. A message passing communication mechanism is proposed to relax the strict synchronization imposed by the simulation-based semantics of VHDL. A partitioning technique is then described which is used to partition the VHDL specification into a hardware part and a software part. The partitioning is carried out during the compilation process of VHDL into a design representation which identifies the hardware/software boundary, while capturing hardware and software in a uniform way to allow efficient co-synthesis of both parts. The VHDL compiler and the partitioning algorithm function as the front end of a hardware/software co-synthesis environment which is built on the design representation.

1. Introduction

We are currently developing a hardware/software co-synthesis environment which accepts a VHDL system-level specification of an application specific system and generates hardware and software implementations which together will implement the given specification. This paper describes two main features of our co-synthesis environment: the use of VHDL for system-level implementation-independent specifications, and a hierarchical, stepwise approach to hardware/software partitioning.

In our environment hardware/software co-synthesis is performed across successive refinement stages. The first stage is carried out on the initial VHDL specification which is *pre-partitioned* into a set of processes that are candidates for hardware and another set for software. This partitioning deals with subprograms, loops, and processes. The resulted VHDL specification is translated into an uniform design representation, providing the possibility to handle, during further design stages, hardware/software trade-offs and re-partitioning at fine granularity. An important characteristics of our approach is that it preserves process interaction semantics during the synthesis process, from the high level VHDL specification to the intermediate design representation for hardware and software, and finally to the synthesized hardware/software implementations.

Several related approaches have been presented for the specification and partitioning of complex hardware/software systems [2], [3], [7]-[9], [11], [12], [16], [17]. They differ in the nature of the initial specifications, the granularity at which hardware/software partitioning is performed, the degree of automation of the partitioning process, and the design step in which partitioning takes place. The hardware-oriented approaches consider the initial description as being a hardware specification; during later design steps parts of the system are decided to be implemented as software. Gupta and De Micheli's approach [8] [9], for example, starts from an initial hardware specification in HardwareC. Portions of the design are later moved into software as long as some design constraints are satisfied.

Systems based on the software-oriented approaches, on the other hand, assume an initial software specification of the system. The Cosyma system [7], for example, accepts a system specification in the form of communicating processes described in an extended version of C. Some parts of the software are then selected for hardware implementation, in order to avoid violation of timing constraints. Similar software-oriented approaches, starting from an initial specification in C (or C++), are presented in [11] and in [1].

The more general approaches start from an implementation-independent specification. One of such approaches is based on an object-oriented functional notation as a co-specification language [17]. In [2] a system for partitioning implementation-independent specifications (based on UNITY) is presented. The use of VHDL as an implementa-

This work has been partially sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK).

0-8186-6315-4/94 $04.00 © 1994 IEEE

tion-independent high level specification is advocated by Ecker [3] and Kumar *et al* [12]. Ecker's approach is based on an early partitioning of hardware and software, which are then separately synthesized, but can be simulated together at different levels of synthesis. Kumar *et al* [12], on the other hand, use an iterative refinement method for hardware/software partitioning.

In our approach we start from an implementation-independent system-level specification in VHDL. Partitioning is then carried out in successive steps, both at coarse and at fine grain level. This stepwise approach has the advantage of reducing complexity of the partitioning process. At the same time it combines the advantages of an early partitioning at the source program level, with those of graph level partitioning. Partitioning at the graph level facilitates accurate estimation of costs and speed, but at the same time makes user interaction practically impossible. Pre-partitioning during the first refinement stage relies mainly on information extracted from the structure of the VHDL specification and on simulation statistics and is performed in interaction with the designer. It results in a readable, back-annotated version of the original VHDL specification. Therefore the designer can easily change the specification to influence the partitioning results. This step is aimed at producing a preparatory hardware/software partitioning so as to reduce the design space for the graph level refinement steps. Later partitioning of the design representation, on the other hand, takes into account the estimated speed and cost in respect to imposed design constraints, and works at the level of operations.

This paper is divided into 5 sections. Section 2 introduces briefly the proposed hardware/software co-synthesis environment. Section 3 discusses features of VHDL as a system level specification language, mainly from the point of view of interprocess communication. In section 4 we focus on the pre-partitioning of the VHDL specification. Finally, section 5 presents the conclusion.

2. The Co-Synthesis Environment

The overall structure of our hardware/software co-synthesis environment is illustrated in Fig. 1. The input specification is given in VHDL which is extended to include a send/receive mechanism for process communication [6]. Such a VHDL program is used to specify only the high-level behavior of the designed system without prescribing the hardware/software boundary or implementation details. After compilation and pre-partitioning, the VHDL specification is translated into a unified design representation which identifies the hardware/software boundary, while capturing hardware and software structures and semantics in a uniform way [15]. This paper concentrates on the VHDL compiler front-end and pre-partitioning package of the co-synthesis environment, while this section briefly presents the basic components of the environment.

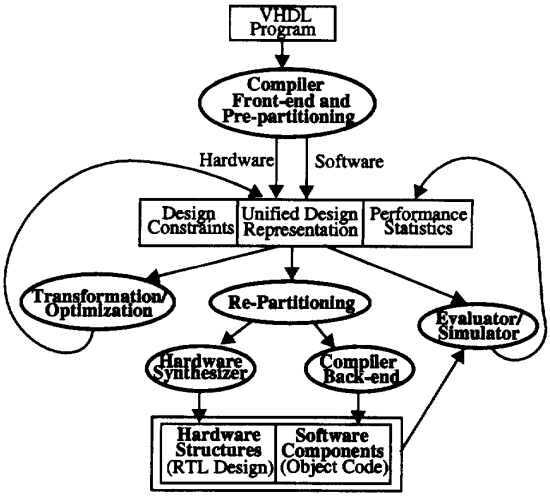

Fig. 1. Overview of the hardware/software co-synthesis environment

The unified design representation is based on an extended timed Petri net notation used originally for hardware modeling [14]. The basic idea is to use Petri nets to represent control flow in both hardware and software, which are extended with a data path representation for hardware operations and a data flow graph representing software operations. The Petri net based notation is also extended with timing information to facilitate performance evaluation.

The compiler front-end and pre-partitioning package of the co-synthesis environment is responsible for compiling the VHDL specifications into the Petri net based design representations. During the compilation process, a set of heuristics is used to identify the *initial* hardware/software boundary. This initial partitioning will eventually be changed to reflect design decisions made to move functionality from one domain to the other so as to increase performance, reduce implementation cost, or satisfy given design constraints during the later synthesis steps. Re-partitioning can also be done by using a graph-based partitioning algorithm as described in [13].

The Petri net based design modeling technique is used to formally capture the intermediate results of design transformations/optimization throughout the *whole* synthesis process. It allows the designer to use verification and evaluation techniques to analyze the intermediate design and make appropriate design trade-offs. One of the evaluation techniques is to use a simulation procedure to *execute* the design representation with typical input stimuli and collect statistics about data usage and control flow choices in a given design. The graph-based partitioning algorithm can then use this performance statistics together with other

design constraints to re-partition the design. When the final partitioning is done, the hardware implementation is synthesized by CAMAD [14], a high-level synthesis system, which is built around a similar design representation. The hardware implementation is considered as a specialized co-processor which will be controlled by and interact with the software generated by a compiler. We assume that the hardware/software implementation architecture consists of a bus controlled by a microprocessor on which the software is also run. Coupled to the bus are a shared memory to be used for hardware/software communication and the implemented hardware.

The main objective of our approach is to develop an integrated set of design tools which are capable of exploring the design space quickly and provide accurate feedback information to the designer. The integration of these tools is achieved by using the extended timed Petri net notation as a unified design representation, coupled with the use of an iterative transformation-based approach to carry out the synthesis process. In this way, the synthesis process is formulated as a sequence of iteration steps which transform an (inefficient) initial design into a final design. The basic concept in this approach is therefore *design transformation*, i.e., to change a design to achieve a goal or meet a constraint. The traditional design tasks such as operation scheduling, resource allocation, and module binding will also be carried out by sequences of transformations. This transformation-based approach to design has also been employed in the high-level synthesis process of the hardware sub-system [14], which results in a uniform approach to handle both hardware/software co-synthesis and hardware synthesis. It is also in line with the operational approach to software development, namely to convert a formal executable specification into an efficient implementation by employing automatic transformations.

3. VHDL for System-Level Specification

Complex digital systems are usually specified as a composition of interacting subsystems, each of them described by a sequential process. VHDL, originally defined as a hardware description language [10], has several features that make it appropriate for system-level, implementation-independent specification. Here are some of the requirements for system-level specification supported by VHDL:

- Data and control abstraction;
- Structural hierarchy;
- Concurrency, synchronization, and communication;
- Timing specification;
- Support for top-down design methodology, covering several levels of the design process;
- Executable specifications (simulation);
- Support for both hardware and software description.

Even though it is defined as a hardware description language, VHDL inherits features of Ada and thus includes constructs appropriate for the description of software. Benefiting from the above-listed features, the designer can specify complex systems in VHDL, without having to consider the possible implementation of one or the other component in hardware or software. It is also possible to execute the specification in order to test the resulted behavior and to collect some statistics that are needed during further design steps. Moreover, the results of successive partitioning and synthesis steps can be specified in VHDL, both for software and for hardware components. There are also pragmatic arguments for using VHDL as the system-level specification language for co-synthesis. For example, there is a great number of commercially available VHDL simulation and hardware synthesis tools which can be integrated into co-synthesis environments.

Accepting VHDL as the input language for a co-synthesis system requires that across all design steps toward the final synthesis result, semantic equivalence with the initial specification and its simulation behavior should be preserved. The main difficulty in this context concerns process interaction. According to the VHDL standard, synchronization and communication between processes are solved using signal assignment and wait statements, the semantics of which is defined in terms of the VHDL simulation cycle. Thus, synthesis of a VHDL design with process interaction specified at the signal level requires practically the implementation of the VHDL simulation cycle in order to achieve semantic correspondence between the initial specification and the synthesized system [6]. However, such a low level synchronization and communication mechanism makes reasoning about processes and their interaction extremely difficult. Therefore, both partitioning and synthesis become extremely complex and inefficient if VHDL signal assignments and wait statements are directly used.

What we need is a high level process interaction mechanism that allows efficient reasoning about processes and their interfaces during partitioning and synthesis, and at the same time can be efficiently implemented both in hardware and in software.

In [4] and [6] we present a model for system-level specification of interacting VHDL processes and describe the hardware synthesis strategy we have developed for it. This model also fits perfectly to the framework of our hardware/software co-synthesis environment. According to the model, processes are the basic modules of the design, and they are interacting using a synchronous message passing mechanism with predefined send/receive commands. Communication channels are represented by VHDL signals. Assignment of a value to a signal is done by a *send* command. Processes that refer to a signal will wait until a value is assigned to it, by calling a *receive* command. Both send

and receive commands have the syntax of ordinary procedure calls in VHDL.

Processes communicating according to this mechanism are loosely coupled and can be implemented without enforcing the strong synchronization implied by the VHDL simulation cycle [6] (such a strong synchronization would be highly inefficient, especially across software/hardware boundaries). At the same time, communication interfaces between processes can be easily established or modified during partitioning, when new processes are created or functionality is moved from one process to another.

In the context of our co-synthesis environment, a VHDL description corresponding to this model can be simulated, partitioned, translated into the unified design representation and then synthesized [6], [15].

4. Pre-partitioning of VHDL System-Level Design

Pre-partitioning is performed in the first stage of the co-synthesis process (Fig. 1). As illustrated in Fig. 2, it takes as input the VHDL system level specification, and generates as output a VHDL model consisting of two sets of interacting processes. The processes in one set are marked as candidates for hardware implementation, while the processes in the other set as software implementation candidates. These two sets of processes are then compiled together by the VHDL front-end compiler to the unified design representation.

Pre-partitioning is performed in three steps:

1. Extraction of loops and subprograms: certain loops and subprograms are identified and extracted from the initial set of processes and are encapsulated as separate processes.
2. Generation and partitioning of the process graph: during this step a partitioning of processes into hardware and software is performed.
3. Merging back some of the processes that have been split in step 1. During step 1 one or several *child* processes are possibly extracted from a *parent* process. If, as the result of step 2, some of the child processes are assigned to the same (hardware or software) partition with their parent process, they are merged back together.

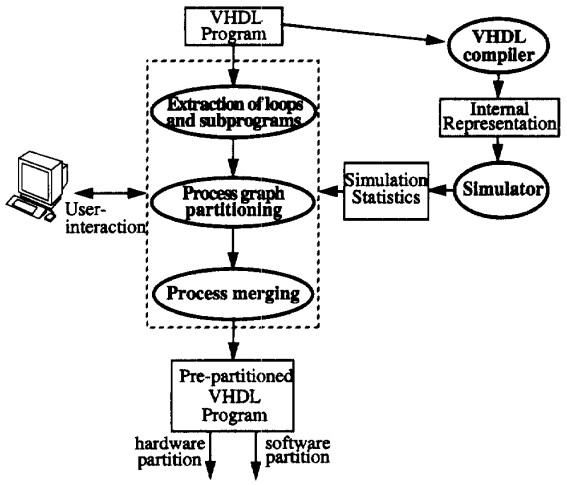

Fig. 2. Pre-partitioning

The first two steps are discussed in more details in sections 4.1 and 4.2, where we will also illustrate that manipulation of processes throughout the whole pre-partitioning process is made possible by the message-passing based interaction model we have developed for system-level VHDL specification.

Our pre-partitioning is strongly based on simulation statistics which are collected by a simulator as depicted in Fig. 2. At the same time, all major decisions taken by the design tools can be influenced through user-interaction. This interaction is straight forward since all operations are performed at the VHDL source level. Simulation, in preparation to pre-partitioning, is carried out with typical input stimuli. Statistics data are collected from simulation of an internal representation generated by our VHDL compiler. Two types of statistics are used for pre-partitioning:

1. *Computation load (CL)* of a basic module (i.e., loop, subprogram, or process) is the total number of operations (at the level of the internal representation) executed by that module, considering all its activations during the simulation. The *relative computation load (RCL)* of a loop or a subprogram is defined as a fraction relative to the computation load of the process the module belongs to; the relative computation load of a process is a fraction relative to the total computation load of the system.
2. *Communication intensity (CI)* on a channel is expressed as the total number of *send* operations executed on the corresponding signal.

4. 1. Extraction of Loops and Subprograms

During the first pre-partitioning step, processes are investigated individually. The purpose is to identify and extract those basic regions that are responsible for most of the execution time spent inside a process (regions that can possibly be considered for hardware implementation). The basic regions are loops and subprograms. The designer guides the identification of regions for extraction, by imposing two boundary values:

1. A lower limit X on the RCL of processes that are investigated for loop and subprogram extraction.
2. A lower limit Y on the RCL of a loop or subprogram to be considered for extraction.

The search for candidate regions in the processes with

RCL greater than X is performed bottom-up, starting from the most inner loops and the subprograms that are not containing loops. Unless a loop or subprogram with RCL greater than Y is found (or the most outer process level has been reached) search is continued upward, into loops that contain loops or subprograms containing loops. When a loop or subprogram that has its RCL greater than Y is found, it will be extracted and a new process will be built to have the functionality of the original loop or subprogram. Optionally, the designer can overrule this decision to move a certain region into a separate process.

Communication between the original process and the extracted process will be built using the send/receive mechanism described in section 3. Since synchronization with send/receive does not affect all processes, but only those involved in the specific communication, the original semantics of the VHDL is preserved [6].

It should also be noted that subprograms are handled in two different ways, depending on if they are called from a single process or from several processes. Subprograms called from several processes are automatically extracted into separate processes (if there is no user option for in-line expansion). This solves the problem of protecting shared subprograms at synthesis [5]. Subprograms called by a single process are considered as part of that process, unless they are extracted as separate processes by the above-described extraction algorithm.

In the following example we illustrate the generation of new processes for a loop and a subprogram that are extracted from the following two processes:

```
P1: process                          P2: process
  ...                                  ...
  LOOP_1: while x<k loop               procedure p(a: in integer;
    ...                                    b: out integer) is
    x := c+k;                          begin
    ...                                  ...
  end loop LOOP_1;                       b := ... a ... ;
  ...                                    ...
end process P1;                        end p;
                                     begin
                                       ...
                                       p(7, z);
                                       ...
                                     end process P2;
```

Given that the RCL's of the loop and the subprogram in the above example are greater than the lower limit Y, two new processes, *P1_LOOP_1* and *P2_PROC_p*, will be generated to execute the operations of loop *LOOP_1 and* subprogram *p* respectively. Communication channels to and from the new processes are established according to the data dependence relationship. In the above example, signals *s_P1_c*, *s_P1_k*, *s_P1_x_to*, and s_*P1_x_from* are introduced for communication between *P1* and *P1_LOOP*_1, and signals *s_P2_a* and *s_P2_b* for communication between *P2* and *P2_PROC_p*. At process generation additional parallelism is also introduced, as far as data dependency allows, by moving statements of the parent process into the sequence between the send and the receive commands used for synchronization with the child process. The new VHDL code after the extraction step is as follows.

```
signal s_P1_c, s_P1_k, s_P1_x_to, s_P1_x_from, s_P2_a, s_P2_b:
  integer;

P1: process                          P2: process
  ...                                  ...
  send(s_P1_c, c, s_P1_x_to,           send(s_P2_a, 7);
    x, s_P1_k, k);                     ... -- additional parallelism
  ... -- additional parallelism        receive(s_P2_b);
  receive(s_P1_x_from);                z := s_P2_b;
  x := s_P1_x_from;                    ...
  ...                                end process P2;
end process P1;
                                     P2_PROC_p: process
P1_LOOP_1: process                     variable b: integer;
  variable x : integer;              begin
begin                                  receive(s_P2_a);
  receive(s_P1_c, s_P1_x_to,           ...
    s_P1_k);                           b := ... s_P2_a ... ;
  x := s_P1_x_to;                      ...
  LOOP_1:while x<s_P1_k loop           send(s_P2_b, b);
    ...                              end process P2_PROC_p;
    x := s_P1_c + s_P1_k;
    ...
  end loop LOOP_1;
  send(s_P1_x_from, x);
end process P1_LOOP_1;
```

The final decision on if processes generated during this step will be kept as separate modules or not depends upon the two subsequent pre-partitioning steps. An important criterion for this decision is the intensity of communication between parent and child processes. It has been ignored at the extraction step, but will be one of the main partitioning criteria considered during the next step.

4. 2. Generation and Partitioning of Process Graph

The VHDL specification resulted from the first pre-partitioning step consists of a set of interacting VHDL processes. Some of these processes are originally specified by the designer; others are generated during loop and subprogram extraction or are subprograms called by more than one process. Statistics concerning computation load of the generated processes and communication intensity on the newly created signals are automatically recomputed during the first pre-partitioning step.

The aim of the second step is to partition the processes into two candidate sets, one for hardware and the other for software. This task is formulated as a graph partitioning problem. We construct a *process graph* where each node corresponds to a process of the new VHDL specification; an edge connects two nodes if and only if there exists a direct communication channel between the corresponding processes (i.e., there exists at least one signal with one process executing send to it and the other receiving from it).

Let us consider the following VHDL example resulted from the first pre-partitioning step.

```
port(ip1, ip2: in integer; op1, op2: out integer);
...
signal s1, s2, s3, s4, s5, s6: integer;
P1: process
    ...
    receive(ip1);
    ...
    send(s1, . . .);
    ...
    send(s3, . . .);
    ...
    receive(s6);
    ...
end process P1;

P2: process
    ...
    receive(ip2);
    ...
    receive(s1);
    ...
    send(op2, . . .);
    ...
end process P2;

P3: process
    ...
    receive(s4);
    ...
    send(s2, . . .);
    ...
end process P3;

P4: process
    ...
    receive(s3);
    ...
    send(s5, . . ., s6, . . .);
    ...
end process P4;

P5: process
    ...
    receive(s1,s5);
    ...
    send(s4, . . .);
    ...
end process P5;

P6: process
    ...
    receive(s2);
    ...
    send(op1, . . .);
    ...
end process P6;
```

Fig. 3 illustrates the processes and their interconnection channels. The corresponding process graph is depicted in Fig. 4, which is then partitioned by a graph partitioning algorithm.

The graph partitioning algorithm takes into account the weights associated to each node and edge, which are used to capture the simulation statistics (computation load, relative computation load, and communication intensity) and information extracted from data-flow analysis of the VHDL processes. The following data extracted by data-flow analysis are captured:

Nr_op_i: total number of operations in the dataflow graph of process i;

$Nr\_kind\_op_i$: number of different operations in process i;

L_path_i: length of the critical path (in terms of data dependency) through process i;

The objective of this pre-partitioning step is to cluster processes with large RCL, high uniformity of operations and high potential of parallelism into the hardware partition and to minimize the amount of communication between the two partitions. Thus, the weight W_i^N assigned to process node i, is calculated by the following formula:

$$W_i^N = M^{CL} \times K_i^{CL} + M^U \times K_i^U + M^P \times K_i^P - M^{SO} \times K_i^{SO};$$

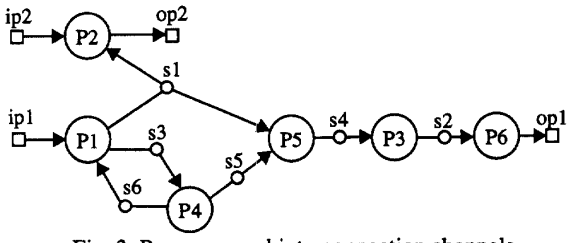

Fig. 3. Processes and interconnection channels

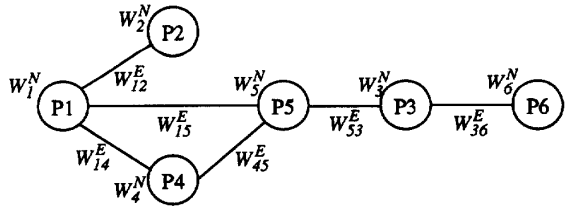

Fig. 4. Process graph example

where:

K_i^{CL} is equal to the RCL of process i, and thus is a measure of the relative computation load of that process;

$K_i^U = \frac{Nr\_op_i}{Nr\_kind\_op_i}$; K_i^U is a measure of the uniformity of operations in process i;

$K_i^P = \frac{Nr\_op_i}{L\_path_i}$; K_i^P is a measure of the potential parallelism inside process i;

$K_i^{SO} = \frac{\sum_{op_j \in SP_i} w_{op_j}}{Nr\_op_i}$; K_i^{SO} captures the extent to which process i contains operations that are most suitable for software implementation. SP_i is the set of such operations (floating point computation, file access, pointer operations, recursive subprogram call, etc.) in process i and w_{op_j} is a weight associated to operation op_j, measuring the degree to which the operation has to be implemented in software; a large weight associated to such an operation will dictate software implementation for the given process, regardless of other criteria.

The relation between the above-named coefficients K^{CL}, K^U, K^P, K^{SO} are regulated by four different weight-multipliers: M^{CL}, M^U, M^P, and M^{SO}, which can be controlled by the designer.

The weight associated to an edge connecting node i and j depends on the amount of communication between process i and j, and is computed by the following formula:

$$W_{ij}^E = \sum_{s_k \in Sig_{ij}} wd_{s_k} \times CI_{s_k};$$

where Sig_{ij} is the set of signals which are used for communication between process i and j; wd_{s_k} is the width (number of bits) of signal s_k; and CI_{s_k} is the communication intensity on signal s_k.

After constructing the process graph and assigning weights to its nodes and edges, the graph is partitioned into a hardware and a software subgraph. A simulated-annealing algorithm, similar to the one presented in [13], has been used for partitioning the weighted graph. The cost function is defined by the sum of weights assigned to cut edges, which is to be minimized under the constraints that no node

with a weight smaller than a given limit W_{Lim1} should be assigned to the hardware subgraph and no node with a weight greater than a limit W_{Lim2} should be assigned to the software subgraph. Limits W_{Lim1} and W_{Lim2} are regulated by the designer.

The two subgraphs resulted after partitioning represent the two set of processes that are considered as candidates for hardware respectively software implementation. They will be compiled to the corresponding design representation for further design steps.

5. Conclusion

We have presented an approach to system-level specification and partitioning in hardware/software co-design. It is based on the basic idea that a hardware/software co-design system should allow designers to start the design process with a behavioral specification which does not prescribe the hardware/software boundary or implementation details. We have shown in this paper that such a specification can be captured by the VHDL language. While arguing the advantages of using VHDL, the main limitation of VHDL as a hardware/software co-specification language has also been identified and a solution to resolve the problem is proposed. Our solution is based on the concept of reduced synchronization between VHDL processes, which is used to relax the strict synchronization imposed by the simulation-based semantics of VHDL. As a result of this, our method preserves semantics correspondence between the system specification in VHDL and its final implementation in hardware and software, with little synchronization overhead and minimal impact on system performance.

Another basic idea of our approach is that design partitioning should be carried out at different stages. Starting from a pre-partitioning at process, subprogram and loop level, re-partitioning will successively be carried out when implementation details are added. Since the pre-partitioning dictates the global design structure, it must be carefully done. In this paper, we present a technique to perform pre-partitioning of the input VHDL specification, which supports the use of simulation results as well as user-interaction to guarantee the partitioning quality. With the formulation of the main partitioning step as a graph partitioning problem based on different cost matrices, our algorithm is very flexible and can be adapted for different partitioning styles and applications. With part of the implementation and experiment work still going on, we expect some modification to the current matrix to take place. Nevertheless, the graph partitioning algorithm we have implemented has already been used successfully in fine-grain partitioning in our hardware/software co-synthesis environment.

References

[1] P. M. Athanas, H. F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *Computer*, March 1993.

[2] E. da Silva Barros, "Hardware/Software Partitioning using UNITY," Ph.D. thesis, Fakultat fur Informatik, Universitat Tubingen, 1993.

[3] W. Ecker, "Using VHDL for HW/SW Co-Specification," *Proc. EURO-DAC/EURO-VHDL'93*, Sept. 1993.

[4] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, "Two Methods for Synthesizing VHDL Concurrent Processes," Research Report, LiTH-IDA-R-93-22, Dept. of Computer and Information Science, Linköping University, 1993.

[5] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, "Synthesis of VHDL Subprograms and Processes in the CAMAD System," *Proc. Workshop on Design Methodologies for Microelectronics and Signal Processing*, Cracow, Poland, Oct. 1993.

[6] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, "Synthesis of VHDL Concurrent Processes," *Proc. EURO-DAC/EURO-VHDL'94*, Sept. 1994.

[7] R. Ernst, J Henkel, T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, Dec. 1993.

[8] R.K. Gupta, G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, Sept. 1993.

[9] R. K. Gupta, C. N. Coelho Jr., G. De Micheli, "Program Implementation Schemes for Hardware-Software Systems," *Computer*, Jan. 1994.

[10] *IEEE Standard VHDL Language Reference*, IEEE Std. 1076-1987, IEEE Computer Society Press, 1987.

[11] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, H. Tenhunen, "A Software Oriented Approach to Hardware/Software Codesign," *Proc. International Conf. on Compiler Construction*, April 1994.

[12] S. Kumar, J. H. Aylor, B. W. Johnson, Wm. A. Wulf, "A Framework for Hardware/Software Codesign," *Computer*, Dec. 1993.

[13] Z. Peng, K. Kuchcinski, "An Algorithm for Partitioning of Application Specific Systems," *Proc. EDAC'93*, March 1993.

[14] Z. Peng, K. Kuchcinski, "Automated Transformation of Algorithms into Register-Transfer Level Implementation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Feb. 1994.

[15] E. Stoy, Z. Peng, "A Design Representation for Hardware/Software Cosynthesis," *Proc. Euromicro Conference'94, System Architecture and Integration*, Sept. 1994.

[16] D. E. Thomas, J. K. Adams, H. Schmit, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, Sept. 1993.

[17] N. S. Woo, A. E. Dunlop, W. Wolf, "Codesign from Cospecification," *Computer*, Jan. 1994.