OPTIMIZED IMPLEMENTATION OF APPLICATION SPECIFIC INTEGRATED CIRCUITS SPECIFIED WITH DEPENDENCE GRAPH

LINDA KAOUANE[1]: PHD STUDENT, MOHAMED AKIL[1], YVES SOREL[2]

[1]Groupe ESIEE–Laboratoire A2SI,
BP 99 - 93162 Noisy-le-Grand, France
E-mails :kaouanel@esiee.fr, akilm@esiee.fr

[2]INRIA Rocquencourt–OSTRE,
BP 105 - 78153 Le Chesnay Cedex, France
E-mail : yves.sorel@inria.fr

1 Introduction

The increasing complexity of signal, image and control processing algorithms in real-time embedded applications requires efficient system-level design methodologies to help the designer to solve the specification, validation and synthesis problems. In order to achieve this goal, the OSTRE team of INRIA has developped the AAA (Algorithm-Architecture Adequation) rapid prototyping methodology which helps the real-time application designer to obtain rapidly an efficient implementation of his application algorithm on his heterogenous multiprocessors architecture and to generate automatically the corresponding distributed executive. This methodology is based on graphs models to exhibit both the potentiel parallelism of the algorithm and the available parallelism of the multicomponent. The implementation is formalized in terms of transformations applied on the previously defined graphs.

Indeed the real-time and embedded constraints may be so strong that the available high performant processors are not sufficient. Which leads to the use, in complement of processors, of specific components like ASIC or FPGA. The aim of our research is to extend the AAA methodology to the implementation of real-time applications on specific integrated circuits. This extension uses a single factorized graph model, from the algorithm specification down to the architecture implementation, through optimizations expressed in terms of defactotization transformations.

2 Algorithm model

Basically, the algorithm model is an extention of the directed acyclic graph of operations (DAG), where each node models an operation, and each oriented hyperedge models a data, produced as output of a node, and used as input of an other node or several other nodes (data diffusion). For example, in image processing, this specification model based on data dependence between implementable operations frequently generates repetitions of operation patterns (identical operations that operate on different data). To reduce the size of the specification and to highlight these regular parts we use a graph factorization process which consists in replacing a repeated pattern, i.e. a subgraph, by only one instance of the pattern, and in marking each edge crossing the pattern frontier with a special “factorization” node. The type of the factorization node depends on the way the data are managed when crossing a factorization frontier (Fork node (F) for array partition, Join node (J) for array composition, Diffusion node (D) for diffusion of a data, Iterate node (I) for data dependence between iterations).

This extention provides specification of loops through factorization nodes and conditioned operations (operation executed or not, depending on its conditioning input) leading to an algorithm model, that we call Conditioned (conditional execution) Factorized (loop) Data Dependences Graph (CFDDG).

3 Relationship between factorization frontiers

Every factorization frontier may be a consumer (located downstream) or/and a producer (located upstream) relatively to another frontier according to the data dependences. Two frontiers are neighbor if there is at least one relation of direct dependence that does not cross a third frontier. We build a graph based on these neighborhood relations where each node represent the factorization frontier and each oriented edge represent the data flow between factorization frontiers.

In the case of a sequential implementation of factorization nodes, every factorization frontier separates two regions, the first one called ”fast”, being repeated relatively to the second one, called ”slow”. Every node of the neighborhood graph is then subdivided in four parts: slow-downstream, fast-upstream, fast-downstream and slow-upstream. This neighborhood graph, deduced automatically from the CFDDG, is used during the implementation to establish the control relationships between frontiers.

4 Circuits synthesis

The transformed algorithm graph, obtained after optimization by defactorization is transformed in a hardware graph containing data and control paths. Data path is deduced by direct translation of each algorithmic node in a hard operator and of each data dependence in a matching physical communication media. The generated control is a local control based on the RTL model that associates to each frontier node of the neighborhood graph a control unit to assume data transfers synchronization. This approach allowed us to carry out an automatic generator of synthesizable structural VHDL.

5 Implementation optimization: principles

For a given algorithm graph, there is a large, but finite, number of possible implementations which are more or less defactorized, and among which we need to select the most efficient one, i.e which satisfies real-time constraints and which uses as less as possible the hardware ressources, logic gates for ASIC and number of Configurable Logic Blocks for FPGA. This optimization problem is known to be NP-hard, this is why we use heuristics guided by their cost function in order to compare the performances of the different defactorizations of the specification. These heuristics, using tricks related to practice, allows us to explore only a small subset of all the possible defactorizations into the implementation space. These cost functions take into account the three characteristics of an implementation: hardware ressources required, latency and data-rate.