A methodology to implement real-time applications on reconfigurable circuits

Linda Kaouane, Mohamed Akil, Thierry Grandpierre
({kaouanel,akilm,grandpit}@esiee.fr)
Groupe ESIEE–Laboratoire A2SI, BP 99 - 93162 Noisy-le-Grand,
Equipe BIOSEM, ESIEE & université Paris12, France.

Yves Sorel (yves.sorel@inria.fr)
INRIA Rocquencourt–OSTRE, BP 105 - 78153 Le Chesnay Cedex, France.

Abstract. This paper presents an extension of the AAA rapid prototyping methodology for the optimized implementation of real-time applications onto reconfigurable circuits. This extension is based on an unified model of factorized data dependence graphs as well to specify the application algorihtm, as to deduce the possible implementations onto reconfigurable hardware. This is formalized in terms of graphs transformations. This seamless transformation flow has been implemented in SynDEx, a system level CAD software tool.

Keywords: rapid prototyping, circuit synthesis, graph transformations, optimizations, heuristics, reconfigurable circuits FPGA

1. Introduction

The increasing complexity of signal, image and control processing in embedded real-time applications requires high computational power to meet real-time constraints. This power can be achieved by high performance mixed hardware architectures, called "multicomponent" [1], built from different types of programmable components (RISC or CISC processors, DSP,..) to perform high level tasks and/or specific non programmable components (dedicated boards, ASIC, FPGA,...) used to perform efficiently low level tasks such as signal and image processing and devices control. Implementing these complex algorithms onto such distributed and heterogeneous architectures while verifying the severe real-time constraints is generally a complex task. This explains the real need for dedicated high level graphical design environments based on efficient system-level design methodologies to help the real-time application designer to solve the specification, validation and synthesis problems [2].

In order to cope with these increasing needs, on the one hand the AAA (Algorithm-Architecture Adequation) rapid prototyping methodology was proposed and the associated software tool SynDEx was developed. AAA/SynDEx helps the real-time application designer to obtain

rapidly an efficient implementation (i.e which meets real-time constraints and minimizes the architecture size) of his application algorithm on his heterogeneous multiprocessors architecture. Finally, SynDEx is able to generate automatically the corresponding distributed executive [1]. This methodology is based on graphs models in order to model the application algorithm, the available multiprocessors architecture as well as the implementation which is formalized in terms of transformations applied on the previous graphs.

On the other hand we aim to extend the AAA methodology to the hardware implementation of real-time applications onto specific integrated circuits, in order to finally provide a general methodogy allowing to automate the implementation of complex application onto multicomponent architecture, these specific integrated circuits beeing also specified in the same framework. This extension uses a single factorized graph model, from the algorithm specification down to the architecture implementation, through optimizations expressed in terms of defactorization transformations [3]. This optimization aims to satisfy the real-time constraints while minimizing the required hardware resources. In prospect, this extension is expected to allow the AAA methodology to be used for optimized hardware/software codesign and consequently to provide generation of either executives for the programmable parts of the architecture (network of processors), or structural synthesizable VHDL for the non-programmable parts (network of application specific circuits and/or FPGA).

This paper presents our extended methodology and is organized as follows. After the related work, we briefly present, in section 3, the transformation flow used by our methodology to automate the hardware implementation process of an application algorithm onto reconfigurable circuits. Then, we present in section 4 the factorized data dependence graph model proposed to specify the application algorithm. Section 5 present an intermediate graph called the neighborhood graph, built to allow the control path synthesis of the implementation. In the next section, a motivating example of matrix-vector product used to illustrate the methodology is described. We then present in section 7 the principles allowing to automate the synthesis of both data and control paths from the algorithm specification. The principles of optimization by defactorization are shown in section 8 while our software tool SynDEx-IC which implement the extended methodology is presented in section 9. In section 10 we show the results of the implementation of the matrix-vector product algorithm onto a *Xilinx* FPGA following the design flow used by the proposed methodology. Finally, section 11 concludes and discusses future work.

2. Related Work

In the field of embedded real-time applications several system-level design methodologies have adressed the issues of design space exploration, performance analysis, mapping and optimizing applications onto different types of hardware architecture.

For example, the POLIS system[1] implements a HW/SW codesign usign the CFSM [4](the Codesign Finite State Machine formal model) but it doesn't support automatic partitioning. An extension of POLIS system by integrating an automated partitioning system is presented in [5]. Working on database of reusable software (C, assembler) and hardware modules (VHDL), the partitioning process passes back the allocation information into POLIS, where a first verification can be performed by Ptolemy[2] based simulation. Finally, the partitioning choice is verified, by using an emulator environment (CPU core coupled with FPGA boards).

GRAPE-II [6] is a system-level development environment for specifying, compiling, debugging, simulating and emulating digital-signal processing applications on heterogenous target plaforms consisting of DSPs and FPGAs. In the specification phase, the application is described using a cycle-static data flow. The application is represented as a directed graph, where nodes represent computation tasks, and edges the communications of the results (tokens). The fonctionality of the nodes is specified in conventional high level language (C, VHDL). The target architecture is specified as a connectivity graph. After specification, resources requirement, mapping architecture, the last phase generates C or VHDL code for each of the processing devices.

The SPADE methodology [7] enables modeling and exploration of heterogeneous signal processing systems onto coarse-grain data-flow architectures. Applications can be structured starting from available C-code using the Khan API functions (the Khan Process Networks model is used to specify the application). SPADE design flow uses trace driven simulation to co-simule an application model with an architecture model.

SPARK [8] is a high-level synthesis framwework that provides a number of code transformations techniques. SPARK takes behavioral ANSI-C code as input and generates synthesizable RTL VHDL. This VHDL can then be synthesized into an ASIC or mapped onto an FPGA (the synthesized control is a finite state machine controller).

Each methodology has its own features: in order to have a complete environment, based upon POLIS, one have to extend it by using

[1] http://www-cad.eecs.berkeley.edu/~polis/
[2] http://ptolemy.eecs.berkeley.edu/

Ptolemy (for cosimulation, visualization) and a partitioning module. Then, the POLIS design flow is heterogeneous. The SPADE methodology adresses a subset of architecture model (coarse-grain data flow) and focusses only on application modeling and simulation. The SPARK high-level synthesis environment is dedicated to system-on-chip plateform. But the presented model seems to be not generic: only some SoC may be supported and it doesn't take into account a heterogeneous architecture (network of different types of components: processors, dedicated circuits (ASIC, FPGA), communication components. Based upon a single design flow, GRAPE-II is used to implement the synchronous DSP applications on heterogeneous target plateforms consisting of DSPs and FPGAs. GRAPE-II methodology is close to the AAA methodology by using a single design flow for specific subset of architecture. The main distinction is that AAA methodology adresses a generic architecture model called multicomponent architecture and gives all formal graph transformations that lead to an optimized implementation. Based upon graphs models, the AAA implementation process consists in distributing and scheduling the algorithm graph onto the multicomponent architecture graph while satisfying the real-time constraints. Basically the AAA/SynDEx for multiprocessors, allows to generate automatically the dead-lock free executive for the optimized implementation of the given algorithm onto architectures based on DSP (TMS320C40, ADSP21060), microcontrollers (MPC555), general purpose processors (linux PC and unix workstations), that are interconnected by various communication medias (shared memories, serial or parallel DSP links, Ethernet,....) [9]. The principles described in this paper aim to extend the AAA/SynDEx for circuits synthesis applied to reconfigurable components (FPGA). Based on the unified model, we can generate both the data and control paths corresponding to hardware implementation using a seamless unified flow of transformations. This work is an intermediate step in order to finally provide a methodogy allowing to automate the optimized hardware/software implementation. Consequently, this extension is expected to allow the AAA methodology to be used for optimized hardware/software codesign.

3. AAA methodology for integrated circuits

Given an algorithm graph G_{al} specifying the application, we transform it into an implementation graph G_{im} following a set of graphs transformations as described in figure 1. This transformation flow is composed of the generation of the data-path graph G_{dp} and the control-path graph G_{cp}. Data-path transformations are quite simple, but control-

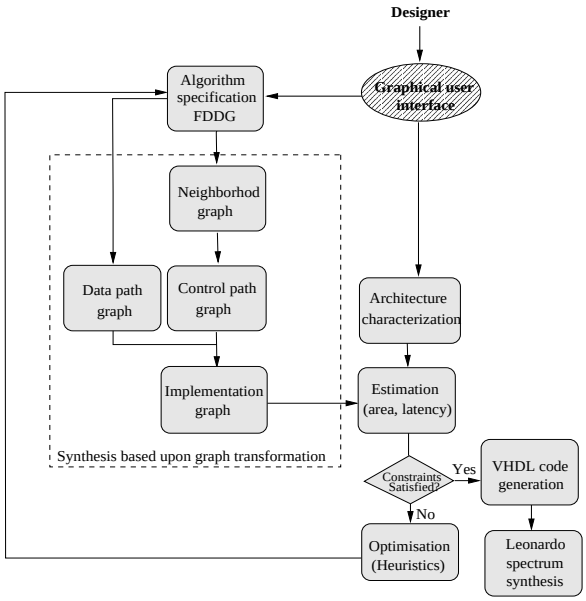


Figure 1. The AAA design flow for circuits

path transformations are not trivial and require to build first a neighborhood graph G_{ng}. Finally the implementation graph ($G_{im} = G_{dp} \cup G_{cp}$) containing both data and control graphs is charaterized in order to estimate time and surface performance of the implementation. If the deduced implementation does not satisfy the constraints specified by the user, we apply a defactorization process in order to reduce the latency by increasing the hardware resources. Since there is a large but finite number of possible defactorized implementations, among which we need to select the most efficient one. This optimization problem is known to be NP-hard, this is why we need to use heuristics guided by their cost function. Finally, the resulting optimized implementation is then used to generate automatically the corresponding VHDL code.

4. Algorithm model

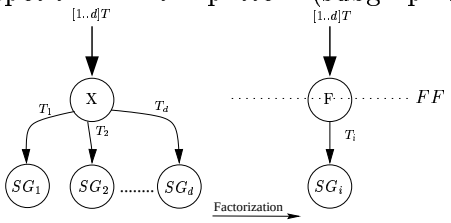
The algorithm specification is the starting point of the process of hardware implementation of an algorithm application onto an architecture. According to the AAA methodology, the algorithm model is an extention of the directed data dependence graph, where each node models an operation (more or less complex, e.g. an addition or a filter), 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 diffu-

sion). In this specifictaion, the order relation between the operations is only determined by their data dependencies, establishing a partial order that exhibits a potential parallelism of the algorithm "potential operation parallelism". Although the purely data dependence model is adequate for expressing the parallelism of computation which it is very attractive for real-time embedded applications, it is not sufficient for expressing repetition inherent in such applications. A more general data dependence model is thus needed. That is why, we extend the typical data dependence model to provide specification of loops through factorization nodes, leading to an algorithm model called Factorized Data Dependence Graph (FDDG). In this model, each dependence is a data dependence and each node is either a computation operation, an input-output operation, or a repetitive operation. We will see in section 9 that this algorithm graph may be specified directly by the user using the graphical interface of the SynDEx software.

4.1. Factorized Data Dependence Graphs Model

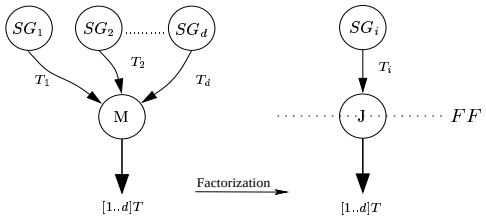
In order to specify his algorithm, the designer frequently has to describe repetitions of operation patterns (identical operations that operate on different data) defining a "potential data parallelism" by opposition to "potential operation parallelisme". To reduce the size of the specification and to highlight these repetitive parts we use in practice a graph factorization process which consists in replacing a repeated pattern, i.e. a subgraph (SG), by only one instance of the pattern, and in marking each edge crossing the pattern frontier with a special "factorization" node. The factorization frontier (FF) is represented by a dotted line crossing these nodes. The type of factorization nodes depends on the way the data are managed when crossing a factorization frontier. Then a factorization node may be:

- **a Fork node (F)**: factorizes array partition by X in as many subarrays as repetitions of the pattern (subgraph SG);



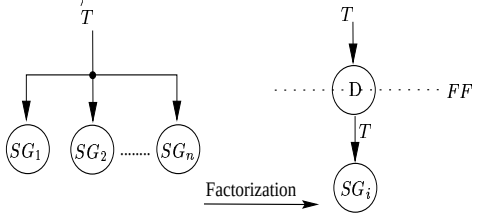
  An infinite fork F^{∞} models an infinite array of inputs from the external environment, it is a graph source that model "Sensor".

- **a Join node (J)**: factorizes array composition by M from results of each repetition of the pattern;

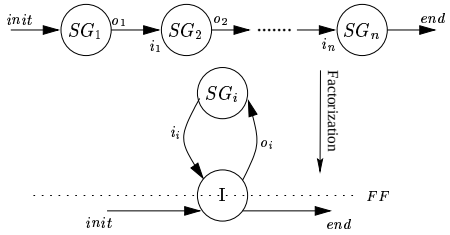


An infinite join J^∞ models an infinite array of outputs to the external environment, it is a graph sink that model "Actuator".

– **a Diffusion node (D)**: factorizes diffusion of a data to all repetitions of the pattern;



– **an Iterate node (I)**: factorizes inter-pattern data dependence between iterations of the pattern. The first of which takes its value from the 'init' input, and the last of which gives value to the last output 'end'.



An infinite iterate I^∞, also usually called "delay", has no *last* output.

Note that, since we deal with reactive applications that interact infinitely with the environment, application algorithms are modeled by an infinitely repeated pattern. This pattern factorization leads to what we usually called a "factorized data-dependence graph". Physical sensors correspond to infinitely repeated acquisition of data. Thus we model the sensor with an infinite fork F^∞ in the factorized graph. Symmetrically, actuators are modeled with an infinite join J^∞. Communications between successive iterations of the infinitely repeated patterns are modeled by an infinite Iterate I^∞.

The graphs in figure 2 gives two specifications examples of the same scalar product SP of two integer vectors M_i and V of dimension 3, the one in figure 2.a is a non factorized data dependence graph and

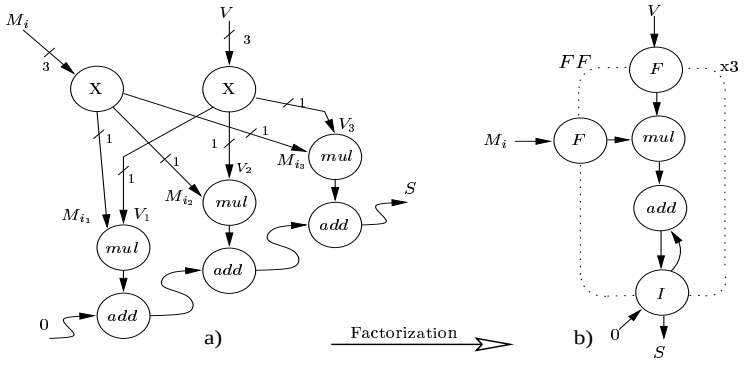


Figure 2. The factorization of a scalar product

the one in figure 2.b is the equivalent (from the specification point of view) factorized data dependence graph. In figure 2.a the nodes X are an array-decomposition operation which separates its input array V (respec. M_i) into its elements. Althought apparently, figure 2.a and figure 2.b are not the same graph (different nodes and edges), they have the same semantics: apply the product operation *mul* as many times (3 times in this example) as there are elements in the vectors to multiply and accumulate the sum. Thus, from the algorithm specification point of view, the factorization reduces only the size of the specification, without any modification of its semantic. However, from the implementation point of view, the factorization allows all the possible implementations, from the all parallel one to the all sequential one, with all the intermediate cases mixing both sequential and parallel. The factorized graph of figure 2.b may be implemented of one of all its possible implementations. That is to say, an implementation where the three multiply operators will be executed sequentially through an iteration, or will be executed all in parallel like in figure 2.a, or two of them will be executed in parallel and executed sequentially with the third one, etc. Obviously, each of these implementation will have different characteristics in terms of area and response time.

5. Neighborhood graph

According to the data dependences relating the factorization frontiers, every factorization frontier may be a consumer (located downstream) or/and a producer (located upstream) relatively to another frontier. Two frontiers are neighbor if there is at least one relation of direct dependence that does not cross a third frontier.

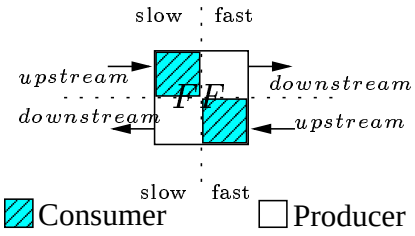


Figure 3. Node of neighborhood graph representing a factorization frontier FF

Based on these neighborhood relations between the factorization frontiers in the algorithm graph G_{al}, we build a neighborhood graph G_{ng}. The nodes of such graph represent the factorization frontiers and the oriented edges represent the data flow between factorization frontiers. The edge orientation describes the consommation/production relation: an edge starts at a producer and ends at a consumer.

In the case of a sequential implementation of factorization nodes, every factorization frontier, called FF, separates two regions, the first one called "fast", being repeated relatively to the second one, called "slow". These slow and fast sides of a frontier are due to the difference of data transfer rate on each side of the factorization frontier. Every node of the neighborhood graph is then subdivided in four parts (see figure 3):

- slow-downstream: "slow" side of a consumer FF;
- fast-upstream: "fast" side of a producer FF;
- fast-downstream: "fast" side of a consumer FF;
- slow-upstream: "slow" side of a producer FF.

This neighborhood graph, deduced automatically from the FDDG, will be used during the implementation (see section 7.2.1) in order to establish the control relationships between frontiers.

6. Example: Specification of (MVP) Matrix-Vector Product

We now use a Matrix-Vector Product example (MVP) to illustrate the algorithm model of specification and how it is used to perform the neighborhood graph. The choice of this example was motivated on the one hand because it presents regular computation on different array data which highlight the use of the factorization process, and on the other hand because it concentrates its computation in nested loops that manipulate multidimensional array data structures and such computations are of interest in signal and image processing applications. So the

MVP of one matrix $M \in R^m \times R^n$ by a vector $V \in R^n$ gives a vector $C \in R^m$, and can be written in a factorized form as follows:

$$C = \left[\sum_{j=1}^{n} m_{ij} v_j \right]_{i=1}^{m} \quad (1)$$

where
m : number of lines of the matrix M,
n : number of columns of M, size of vector V,
m_{ij} : i-j-th element of the matrix M,
v_j : jth element of the vector V.

Equation 1 allows us to obtain the graph corresponding to the algorithm specification of the factorized MVP (figure 4). The interface with the physical environment is delimited by input (F_M^∞ et F_V^∞) and by output (J_C^∞). It corresponds to the factorization frontier of the infinitely repeated pattern of the graph (FF_1) due to the reactive aspect of embedded real-time applications. Indeed, these applications interact infinitely with the physical environment by consuming data provided by sensors and producing data through actuators. The output data are the result of operations applied on the input data. The square brackets $[\ \ ]_{i=1}^{m}$ correspond to a second frontier (FF_2), delimited by factorization nodes of a finitely repeated pattern. This frontier selects the m lines of the matrix M (F_{21}), diffuses the vector V (D_{21}) and collects the result vector C (J_{21}). The functor $\sum_{j=1}^{n}$ corresponds to a third frontier (FF_3), also delimited by factorization nodes of a second finitely repeated pattern corresponding to the calculation of the scalar product M_iV. This frontier selects the m_{ij} elements of the ith line of the matrix M (F_{31}) and the elements v_j of the vector V (F_{32}) and it supplies the result of the sum of products between m_{ij} and v_j for every line of matrix M (I_{31}). The "slow" and "fast" sides of each frontier are labeled "s" and "f", respectively.

The neighborhood graph between factorization frontiers, obtained from the factorized data dependence graph specifying the MVP algorithm, is shown by the figure 5. Because the factorization frontier FF_1 is infinite, it does not have neighbor on its "slow" side which corresponds to the physical environment. FF_1 is, at the same time, a producer (edges M and V) and a consumer (edge C) compared to FF_2. FF_2 is also a producer (edges M_i and V) and a consumer (edge C_i) compared to FF_3. FF_3 is a producer and a consumer, compared to itself through the arithmetic operations *mul* and *add*.

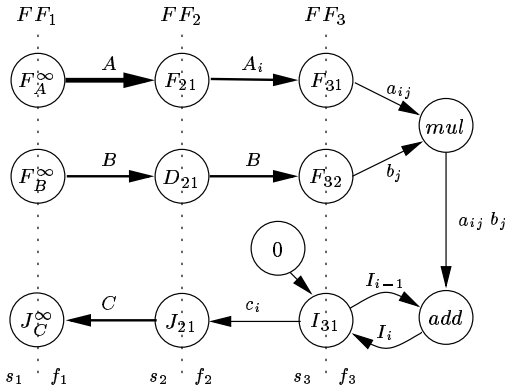


Figure 4. Factorized data dependence graph of MVP

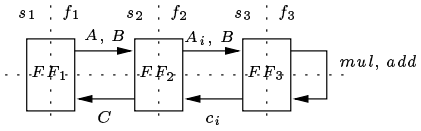


Figure 5. Neighborhood graph of MVP: relations between frontiers

7. Circuits synthesis

To implement the application algorithm on the corresponding circuit we need to generate the data path responsible for the core of the computation as well as control path to generate the appropriate control signals. This translation process from a high-level behavioral representation into a register-transfer-level structural description (RTL) containing both the data and control paths, is known as high-level circuit synthesis. The automation of this synthesis process reduces significantly the development cycle of the circuit, and allows the exploration of different hardware implementations, seeking for an ideal compromise between the area and the response time of the circuit. Afterwards, we will present principles allowing to generate automatically the data path and the control path of the circuit, from the factorized data dependence graph and the neighborhood graph. section 8.1 will explain how to generate an optimized implementation.

7.1. Data path synthesis

The hardware implementation of the factorized data dependence graph consists in providing a matching operator for every operation node and every factorization node. The matching operator is a logic function

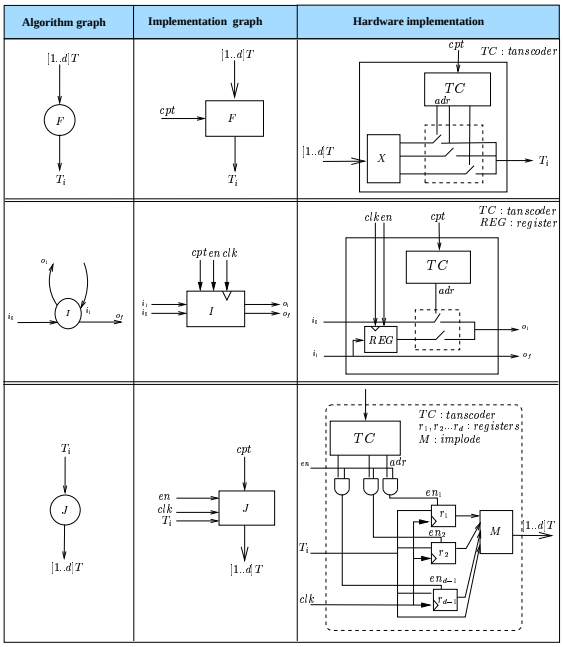


Figure 6. A node graph transformation: from algorithm graph to hardware implementation

in the case of an operation node, or it is composed of a multiplexer and/or registers in the case of a factorization node as depicted in figure 6. Then hardware implementation of the data dependencies between operations consists in providing, for each edge of the graph, a matching connection between operators. The resulted graph of operators and their interconnections compose the data path of the circuit.

7.2. Control path synthesis

The control path corresponds to the logic functions that must be added to the data path, in order to control the multiplexers and the transitions of the registers composing the operators. It is then obtained by synchronization of data transfer between registers. However, two conditions must be satisfied in order to allow a register to change its state: the new upstream data to the register must be stable, and all downstream consumers of the register must have finished the utilization of previous data. Moreover, if upstream data comes from various producers with different propagation time, it is necessary to use a synchronized data transfer process. This synchronization is possible through the use of a request/acknowledge communication protocol [10]. Consequently, the synchronization of the circuit implementing the whole algorithm is re-

duced to the synchronization of the request/acknowledge signals of the set of factorization operators. Given that these operators are gathered in factorization frontier and their data consumptions and productions are done in a synchronous way at the level of the frontier, the generated control must be a local control at each frontier. We propose then a local control system where each factorization frontier will have its own control unit. This delocalized control approach allows the CAD tools used for the synthesis to place the control units closer to the operators to control rather than a centralized control approach : this will minimize classical routing overhead.

7.2.1. *Control units and their interconnections*
As mentioned in section 5, each factorization frontier has upstream and downstream relations on both sides,"slow" and "fast". The relations between upstream/downstream and request/acknowledge signals on both sides of a frontier are implemented by the "control unit" of the factorization frontier. This control unit, depict in figure 7, contains a counter C with d states (corresponding to the d factorized repetitions) and an additional logic function in order to generate, in the one hand the communication protocol between frontiers (the slow/fast, request/acknowledge signals at the upstream and downstream sides), and in the other hand the counter value (cpt) and the enable signal (en), that control the frontier operators. The counter value (cpt) controls the multiplexers of the frontier operators: F, J and I. The enable signal (en) determines the clock cycles where the registers of the frontier operators (F^{∞}, J^{∞}, J and I) will change state. Note that, the signal ($init$) resets the counter while the signal (end) indicates that the counter is in its last state ($d-1$).

All the other signals are the request (r) and acknowledge (a) signals generated by the frontier(s) located upstream or diffused to the frontier(s) located downstream. They are separated in two groups: those which relate to the frontier(s) located on the "slow" side and those which relate to the frontier(s) located on the "fast" side, corresponding to the four parts of the control unit: slow-upstream (su), slow-downstream (sd), fast-upstream (fu) and fast-downstream (fd).

As mentioned above, the control path is mainly composed of the set of control units associated to the factorization frontiers of the application algorithm graph. These control units can then be inter-connected in an automatic way based on relationships between the factorization frontiers deduced from the neighborhood graph. In this control graph, the nodes correspond to the control units and the arcs correspond to the request signals transmitted between the control units in the same way as the production and consumption of data between the corresponding

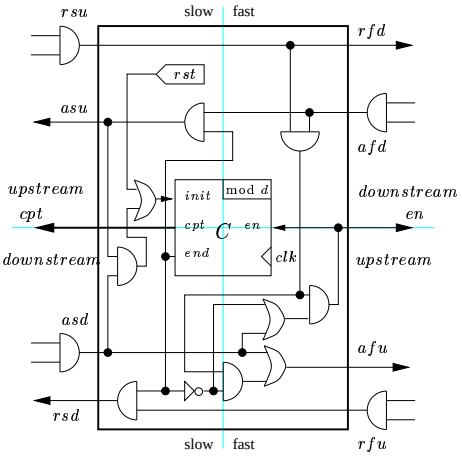


Figure 7. Control Unit

factorization frontiers. The acknowledge signals are transmitted, in the opposite direction of the associated request signals, between the same control units. When several signals arrive at the same input of a control unit, one takes the conjunction by a logical AND. section 10, will present two examples of synthesis of the data and control paths.

8. Implementation optimization

If the implementation of the factorized specification onto an application specific integrated circuit (or onto an FPGA) does not meet the real time constraints, we need to defactorize the implementation graph corresponding to the specification. The defactorization process is the reverse transformation of the factorization and therefore it does not change the operational semantic of the data dependence graph as explained in section 4. The goal is to obtain a more parallel implementation in order to reduce the latency and improve the temporal performances in spite of increasing hardware resources.

Thus the optimized implementation of a factorized algorithm graph onto the target architecture is formalized in terms of graph defactorization transformation. The implementation space which must be explored in order to find the best solution, is then composed of all the possible defactorizations of a factorized graph specifying the algorithm. For instance, for a given algorithm graph with n frontiers, we have at least 2^n defactorized implementations. Moreover, each frontier can be partially defactorized: a factorization frontier of r repetitions can be decomposed in r' factorization frontiers of r/r' repetitions.

Consequently, 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 the real-time constraints (upper bound on latency), and which uses as less as possible the hardware resources (number of logic gates for ASIC and number of Configurable Logic Blocks CLB for FPGA). This optimization problem is known to be NP-hard, and its size is usually huge for realistic applications. This is why we use heuristics guided by a cost function, in order to compare the performances of different defactorizations of the specification. These heuristics allow us to explore only a small but most interresting subset of all the possible defactorizations into the implementation space.

Since we aim rapid prototyping, our heuristic is based on a fast but efficient greedy algorithm, with a cost function f based on the critical path length metric of the implementation graph: it takes into account both the latency T and the area A of the implemantation which are obtained by a preliminary step of characterization.

8.1. Optimization heuristic

Here is a brief description of the proposed greedy iterative heuristic described by the algorithm 1. Note that we are also experimenting other kind of heuristics (simulated annealing) but this not the aim of this paper. At each iteration of the greedy heuristic, a list of candidate factorization frontiers FF_{list} is built from the set of factorization frontiers of the deduced implementation graph G_{im}. These frontiers are those which belong to the critical path CP (line 3 on algo 1). Defactorizing one of these frontiers will reduce the critical path length in order to to meet the real time constraint C_t. Thus for each frontier $FF \in FF_{list}$ (line 4 on algo 1) defactorization factor df_{FF} as the smallest factor of factorization implying a global latency lower than the time constraint C_t(as described in algo 2). In the case of this factor corresponds to the factor of factorization (total defactorization) without a global latency being lower than the time constraint, then this fully defactorized factorization frontier will not be crossed any more in the next critical path computation.

Then we compute for each couple (factorization frontier FF, optimal corresponding factor df_{FF}) the cost function f, called defactorization pressure, as follow:

$$f = \frac{\Delta A}{T - max\{T', C_t\}}$$

where ΔA represent the loss on the area, T the latency before defactorization, T' the latency after defactorization and C_t is the user specified

time contraint. At the end of each iteration, the factorization frontier having the highest cost value will be defactorized by its corresponding df_{FF}.

Algorithm 1 Greedy optimization algorithm

Inputs: The FDD graph G_{FDD}, time constraint C_t
Output: The optimized implementation graph

1: **begin**
2: **while** the latency of the corresponding implementation graph G_{im} is greater than the time constraint C_t **do**
3: compute the critical path CP;
4: determine the list of candidates frontiers FF_{list}: $FF_{list} := \{FF_i, FF_i \in CP\}$
5: **for all** candidate frontier $FF \in FF_{list}$ **do**
6: determine the optimal factor of defactorization df_{FF} as explained in algorithm 2;
7: compute its corresponding cost function f;
8: **end for**
9: defactorize the frontier having the highest cost 'f' by its corresponding df_{FF};
10: **end while**
11: **end**

Algorithm 2 Optimal factor of defactorization algorithm

Inputs: time constraint C_t, current frontiere FF, graph G_{FDD}
Output: Optimal factor of defactorization df_{FF}

1: **begin**
2: df_{FF}:= initial_factorisation_defactor(FF)
3: $G'_{FDD} := G_{FDD}$
4: **while** (latency $(G'_{FDD}) \geq C_t$) and ($df_{FF} \leq$ factor (FF)) **do**
5: $df_{FF} := df_{FF} + 1$;
6: $G'_{FDD} := G'_{FDD}$ with FF defactorized by df_{FF}
7: **end while**
8: **end**

9. SynDEx software tool

The AAA methodology for multicomponent is implemented in the system level CAD software SynDEx[3] (Synchronized Distributed EX-

[3] http://www-rocq.inria.fr/syndex

ecutive). Its graphical user interface enable the user to specify both the algorithm and the architecture graphs. The heuristics of SynDEx provide a distribution and a scheduling of the algorithm operations onto the architecture specified as a graph of processors communicating through a network and/or shared memory [1]. High level dead-lock free executives are then automatically generated. Real-time distributed executive libraries have been developed for networks based on DSP (TMS320C40, ADSP21060), microcontrollers (MPC555), and general purpose processors (linux PC and unix workstations).

The principles described in this paper allowed us to extend the AAA methodology and SynDEx for reconfigurable circuits (FPGA) : SynDEx-IC [4] is the name of the extended tool which support reconfigurable circuit synthesis. The defactorization heuristic and an automatic generator of structural synthesizable VHDL for mono-FPGA (one FPGA) architectures are implemented in SynDEx-IC [11]. The generated VHDL code which corresponds to the optimized FPGA implementation obtained by successive defactorizations of the conditioned factorized algorithm graph, is then used by a CAD tool (e.g. *Leonardo Spectrum*) in order to simulate the design and to generate the netlist needed for FPGA configuration.

10. Example: Synthesis of MVP Implementation on FPGA's circuit

We illustrate now the proposed design flow summarized in figure.1 for the hardware implementation onto FPGA of the MVP example given in section 6. figure.10 shows a snapshot of the MVP algorithm graph specified by the designer using SynDEx-IC. In this hierarchical top-down graphical specification each box represents an operation and each edge data dependencies between operations. The "inMat" and "inVect" boxes in the top left window are sensors that provide input matrix and vector elements. The "outVec" box is the actuator that display the computed value. Top right window specifies the scalar vector product "dotprod" operation. This operation is hierarchically detailed throw the two other windows: "dpacc" box for the multiplication accumulation computation, and the "mul", "add" boxes. figure 10 represents the hardware implementation of the factorized MVP corresponding to the algorithm specification given in figure 4 for $m = n = 6$. The data path (figure 10.a) is composed of the factorization frontier operators ($F_{i,j}$, $D_{i,j}$, $J_{i,j}$ and $I_{i,j}$) and the combinatorial operators *mul* and *add*. The

[4] http://www.esiee.fr/a2si/syndex-ic

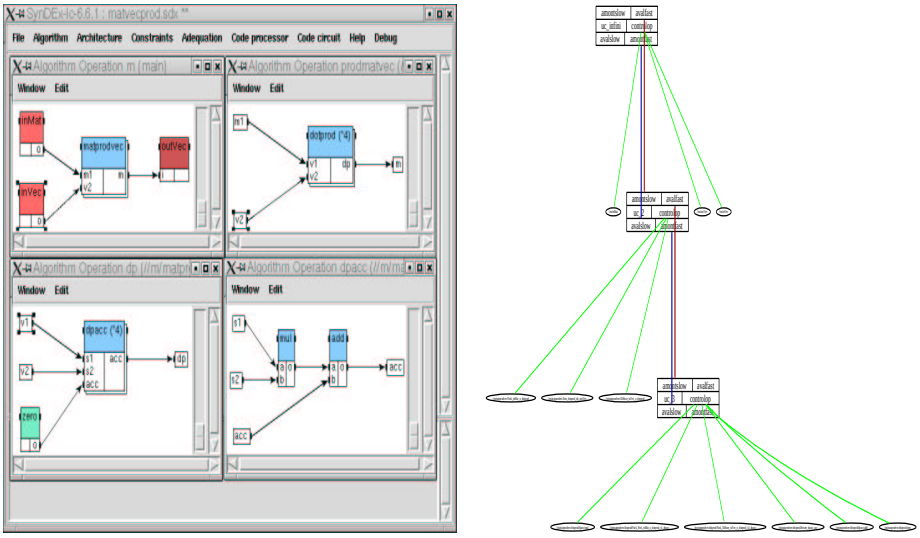


Figure 8. A snapshot of SynDEx-IC screens: algorithm and neighborhood graphs

control path (figure 10.b) is composed of the control units UC_1, UC_2 and UC_3, and of the control signals r(request), a(acknowldge), *cpt* and *en*. The interconnections between the request and acknowledge signals, is based on the relationships between the factorization frontiers, namely the neiborhood graph (figure 5) built from the algorithm graph.

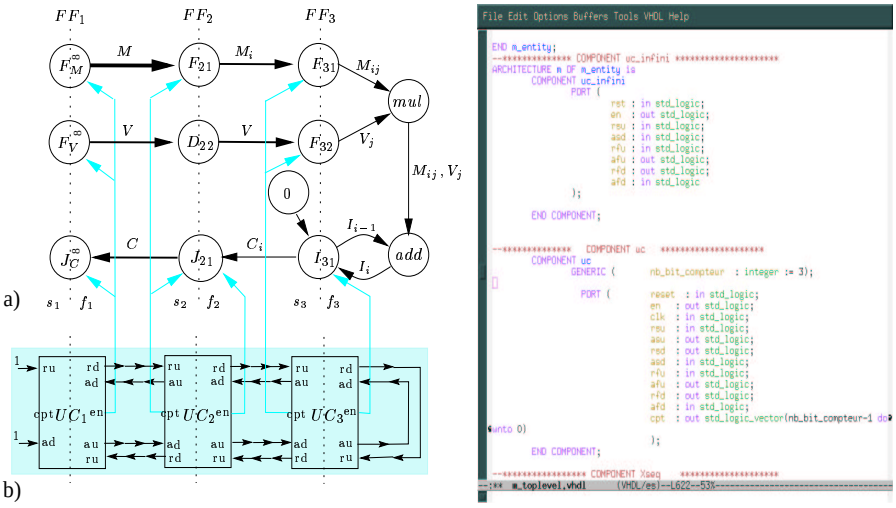


Figure 9. The implementation graph of MVP and the generated VHDL code

In figure 10.a we present the hardware implementation of a defactorized solution corresponding to the partial defactorization of the frontier FF_2 by a factor of 2. The $FF2$ frontier has been replaced by two frontiers FF_{2a}, FF_{2b}, each being repeated 3 times. The factorization frontier FF_3 remains unchanged but it has been duplicated (FF_{3a}, FF_{3b}) due to the partial defactorization of FF_2. The data path is then composed of the factorization frontier operators, the combinatorial operators (mul, add) and of the operators X (array-decomposition operation), M (array-composition operation). The control path, deduced automatically from the neiborhood graph (figure 10.b), is composed of the control units UC_1, UC_{2a},UC_{2b},UC_{3a} and UC_{3b}. The synchronisation of frontiers FF_{2a}, FF_{2b} is assured by the AND gates at the upstream request and the downstream acknowledge of UC_1.

Tab.I shows the implementation results of hardware implementation of MVP (6×6 matrix and 6 elements vector, coded on 3 bits) onto a *Xilinx* FPGA XC4000XL-3, using the CAD tool *Leonardo Spectrum*, developed by *Exemplar Logic Inc.* The implementation results are presented in function of, the area (hardware ressources: number of CLBs), the number of clock cycles required by the algorithm execution, the maximum frequency of operators in *MHz*, and finally the data latency in *ns* (nano seconds).

Table I. Optimization results for the implementation of MVP onto FPGA

Implementation	*Area (CLB)*	*Nb. cycl.*	*Freq. (MHz)*	*Lat. (ns)*
Factorized Spec.	76	36	12,4	2916
Part.defac. by FF_2	99	18	13,5	1332
Fully. defac. by FF_2	168	6	14,3	420
Part. defac. by FF_3	92	30	10,8	2790
Fully. defac. by FF_3	79	6	9,0	660
Fully. defactorized	234	1	11,4	87

These results represent some possible implementations explored by the optimization heuristic by partial defactorization (as described in [3]) of the initial factorized implementation. Note that these defactor-

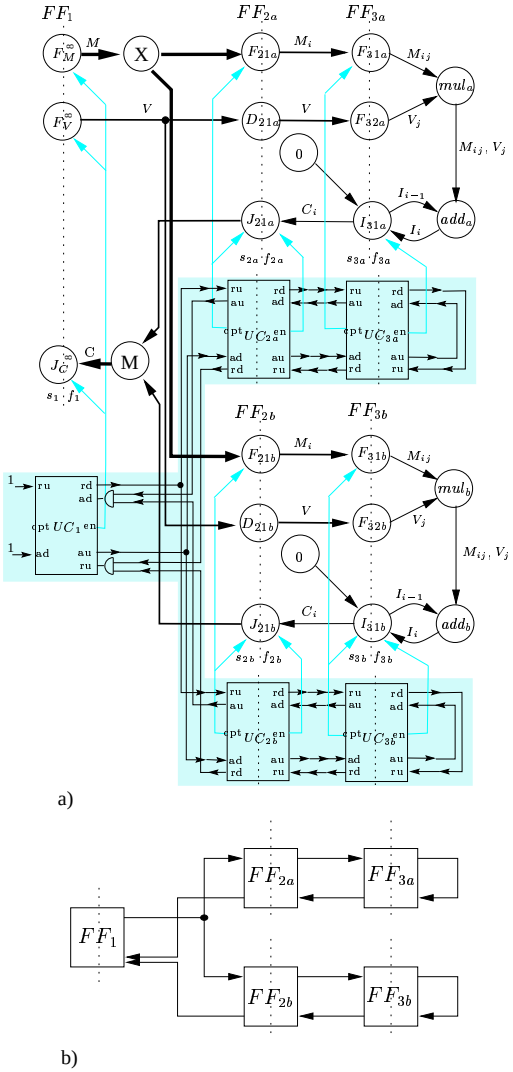


Figure 10. A defactorized implementation graph of MVP

ized solutions allow to reduce the latency of the implementation, but they increase the number of required hardware ressources (CLB).

11. Conclusion and future works

We have presented a seamless flow of transformations that lead to the generation of a complete VHDL design corresponding to the optimized implementation of an application specified by Factorized Data Dependence Graph model.

The principles described in this paper allowed us to extend the AAA/SynDEx for reconfigurable circuits (FPGA) to a new tool named SynDEx-IC. SynDEx-IC is able to automatically generate optimized synthesizable VHDL for mono-FPGA (one FPGA) architectures. The generated VHDL code which corresponds to the optimized FPGA implementation obtained automatically by successive defactorizations of the factorized algorithm graph, is then used by a CAD tool (e.g. *Leonardo Spectrum*) in order to generate the netlist needed for the FPGA configuration.

Presently we are working on the control involved by the conditioning in the algorithm specification, in addition to the control involved by repetition of operation. We intend to extend the proposed methodology to the case of multi-FPGAs architectures. To support such architectures, the optimization heuristic will adress both defactorization and partitioning issues.

Thanks to this extension, the AAA methodology will be used for optimized hardware/software codesign, leading to the generation of either executives for the programmable parts of the architecture (network of processors), or structural synthesizable VHDL for the non-programmable parts (network of application specific circuits and/or FPGA).

References

1. T. Grandpierre, C. Lavarenne, Y. Sorel. *Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors.* CODES'99 7th Intl. Workshop on Hardware/Software Co-Design, Rome, May 1999.
2. S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli.*Design of embedded systems: formal models, validation, and synthesis.* Proceedings of IEEE, v.85, n.3, March 1997.
3. A. F. Dias, C. Lavarenne, M. Akil, Y. Sorel. *Optimized implementation of real-time image processing algorithms on field programmable gate arrays.* Proc. of the 4th Intl. Conference on Signal Processing, Beijing, Oct. 1998.
4. I.D Bates, E.G Chester, D.J Kinniment. *A statechart based HW/SW Codesign system.* Proceedings of the 7 Intl. Workshop on Hardware/Software Codesign (CODES/CASHE), Rome, Italy, 3-5 May 1999.
5. M. Meerwein, C. Baumgartner, W. Glauert. *Linking Codeisgn and Reuse in Embedded Systems Design.* Proceeding of the 8 Intl Workshop on Hardware/Software Codesign (CODES/CASHE), San Diego, California, USA, 3-5 May 2000.
6. R. lauwereins, M. Engels, M. Ad, J. Peperstraete. *Grape-II : A system-level Prototyping Environment For DSP applications.* IEEE Computer, Vol. 28, No 2, pp. 35-43, Feb. 1995.
7. P. Lieverse, P. van detr Wolf, Ed Deprettere, K. Vissers *A Methodology for architecture exploration of heterogeneous signal processing systems.* Proc. 1999 IEEE Worshop on Signal Processing Systems (SiP'99).

8. S. Gupta, N. Dutt, R. Gupta, A. Nicolau *SPARK, High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations.* Intl. Conf. on VLSI Design, January 2003, Mumbai, India.
9. T. Grandpierre, Y. Sorel, *From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations.* First ACM & IEEE Intl. Conference on formal methods and models for codesign. MEMOCODE'03, Mont Saint-Michel, France, june 2003.
10. C. A. Mead, L. A. Conway. *Introduction to VLSI systems.* s.l.: Ed. Addison-Wesley, 1980.
11. R. Vodisek, M. Akil, S.Gailhard, A.Zemva *Automatic Generation of VHDL code for SynDEx v6 software.* Electro technical and Computer Science conference, Portoroz, Slovenia, september 2001.