MULTIPLE ABSTRACTION VIEWS OF FPGA TO MAP PARALLEL APPLICATIONS

*Sébasten Le Beux** *Philippe Marquet* *Jean-Luc Dekeyser*

LIFL and INRIA-Futurs
University of Lille
France

ABSTRACT

Manipulating configurable resources like FPGAs in a co-design framework has become essential: especially, FPGAs may efficiently implement parallel systematic signal processing tasks. Nevertheless, such implementations are usually hand written at a low level. Our proposition is to provide high level modeling of an application and tools to automatically generate tuned VHDL code from these high level models. This paper introduces a flow able to fit a parallel application onto an FPGA according to the FPGA characteristics, and to map this application onto the FPGA. Each step of the flow requires specific details of the FPGA that realize the implementation. To get these details, several views of a same FPGA are introduced: BLACK BOX, QUANTITATIVE *and* PHYSICAL. *The flow automatically generates the VHDL code of the initial application and a constraint file that guides the synthesis tools.*

I. INTRODUCTION

Current Systems on Chip (SoCs) are heterogeneous and integrate computing, storage, communication and interface resources. The latest generation SoCs often include reconfigurable resources, such as FPGAs, providing flexibility. An FPGA allows the realization of a computing, storage or a communication resource. Manipulating these reconfigurable resources in a SoC design framework has become essential and specific methodologies are proposed (see [1] for example).

The Gaspard co-design framework [2] is more specifically oriented towards the co-design of parallel software and hardware. It identifies the parallelism included in regular constructions such as application loops or repetitive constructions of hardware elements. Gaspard, in its first version, is able to program processor based architectures, but does not allow configuration of reconfigurable resources. In this paper, we propose a Gaspard extension allowing the configuration of FPGAs.

The main challenge of this extension is to use the same description when targeting a processor or an FPGA based architecture. When targeting a processor based architecture, tasks are scheduled and executed on the processor. However, when targeting an FPGA based architecture, an hardware implementation of the application onto the FPGA is realized. The required knowledge of the FPGA characteristics depends on the required implementation precision: few details for a first rought design, more details for a tuned RTL design and even more details for a tuned FPGA mapping. Therefore, we identify three different views for the same FPGA, that are associated to the desired implementation precision: BLACK BOX, QUANTITATIVE and PHYSICAL. The aim of this paper is to present our design flow and the associated FPGA views.

The paper is organized as follows. The next section presents a state of the art for loop transformation methodologies that target a hardware execution and presents some FPGA mapping algorithms. This way our modeling of applications allows a compact expression of parallelism and repetition is introduced in Section III. Section IV presents a flow that allows FPGA manipulation in the Gaspard co-design framework. Section V introduces the multiple FPGA abstraction views used in the flow. Finally, Section VI concludes this work and gives perspectives.

II. RELATED WORKS

Several related works which aim at optimizing an hardware implementation of loop based applications have been proposed.

In [3], Dias *et. al.* transform a factorized graph, which expresses the repetition of a task in a compact way. An implementation is obtained by successive defactorizations of the factorized specifications. An heuristic based on a cost function that depends on the consuming resources, the data-rate and the latency, automatically chooses the most efficient defactorized solution. In [4], Kaouane *et. al.* extend this work and automatically generate an hardware implementation. Delocalized control units, placed close to the controlled data-path, schedule the execution according to synchronization frontiers. However, while the defactorization transformation is available, the factorization is not.

Devos *et. al.* describe an extension of CLoog (Chunky Loop Generator) [5], a tool that statically schedules the

*contact author: `Sebastien.Le-Beux@lifl.fr`

execution of an application according to its polyhedral model [6]. The extension partially generates the VHDL that describes the hardware controller and the connection with the execution units, improving data locality. The described extension does not take the implementation constraints into account.

Guillou *et. al.* manage multi-dimensional scheduled uniform recurrence equations with a dedicated controller [7], in the context of the MMAlpha environment [8]. The controller, generated from a polyhedron scanning method, performs data transactions from local memories and generates enable signals that control the computation.

Shesha Shayee *et. al.* take into account IO memory accesses provided by an FPGA and its available area in order to explore implementation solutions [9], [10]. An implementation solution is the result of a sequence of loop transformations. The provided estimations allow compilers to quickly select a feasible and high-performance FPGA design.

Moeller [11] demonstrates that VHDL synthesis and place and route tools do not optimally place a systolic structure onto FPGA. With a manually constraint placement, Moeller enhances the FPGA implementation.

So *et. al.* [12], [13] describe a compiler algorithm that guides hardware design space exploration according to the required computation and the data memory accesses. Transformations are automatically performed, allowing designers to rapidly explore design space. From the initial C code, a VHDL code is generated.

Our proposition differs from these researches in several aspects. Our application modeling takes into account whole applications and not only some academic nested loops. Our application modeling, which is based on ARRAY-OL and its extension [14], is oriented towards the description of systematic signal processing applications. It particularly includes powerful construction that allows to denote all the potential parallelism of an application: data dependencies are explicitly given by the way of high level constructions. Rough data dependencies between arrays will lead to a task parallelism execution while fine grain data dependencies between array elements will lead to a data-parallel execution. Compared with others, our models includes multi-dimensional arrays and our tools include refactoring of arrays that are able to identify temporal and spacial dimensions in arrays.

Others works present FPGA mapping techniques that consider an abstract view of an FPGA. Handa *et. al.* [15] present a fast algorithm that finds unoccupied area on the FPGA. Tasks are placed in a rectangular manner and FPGA is modeled as a homogeneous two dimensional array of CLB. Tabero *et. al.* [16] model the FPGA as a homogeneous three dimensional array, two dimensions for the space and one for the time. The temporal dimension minimizes the area fragmentation and increases the solution search space. Tessier [17] decomposes a FPGA into an array of placement bins. Inside bins, hard macros are first placed, followed by a placement of the more flexible soft macros. According to a placement cost function, Tessier provides an highly efficient placement for most cases, while some placement refinements are necessary for some others cases.

III. CURRENT GASPARD FRAMEWORK

The Gaspard co-design framework and its main concepts are introduced. We first describe the expression of repetitions in Gaspard. Then we provide an example.

III-A. ARRAY-OL: Expression of Repetitions

ARRAY-OL [18], [14] (Array-Oriented Language) is a language specialized in the description of systematic signal processing applications. These kind of applications are characterized by a huge number of data manipulated by a set of regular tasks. ARRAY-OL allows the description of the *task parallelism* and the *data parallelism* that compose an application. The *task parallelism* expresses dependencies between tasks contained in the application, providing a structural aspect of the application. Each task is described using *data parallelism* in the form of a set of inputs and outputs *patterns* consumed and produced by repeated applications of the task. These sets of *patterns* tile the input and output arrays of the task. The *patterns* are defined through an *origin* vector, a *paving* and a *fitting* matrix. The *origin* defines the address in the array of a first *pattern*. The *fitting* defines the shape of the *patterns*. The *paving* defines the iteration of the patterns on the arrays. A formal description of the tilers and some examples are provided in [14]. Elementary tasks are black boxes, such as IP, consuming input *patterns* to produce output *patterns*. Elementary tasks represent atomic computations executed in ARRAY-OL.

Gaspard uses the ARRAY-OL concepts in order to express parallelism. Moreover, some ARRAY-OL extension have been proposed and are also managed in the Gaspard framework. From the ARRAY-OL concepts and its extensions, Gaspard generates different programming languages that allow simulation, execution or synthesis of an application mapped onto an architecture. By generating synchronous languages (Lustre [19] or Signal [20]), it is possible to validate a design application and to detect, for example, dead locks. The automatic generation of procedural languages (e.g. Fortran and C languages) makes possible the execution of concurrent processes onto multiprocessor architectures. The SystemC language allows the simulations at different abstraction level of both application and architecture. The VHDL language makes possible the synthesis. We refer the reader to [21] for detailed presentations and demos.

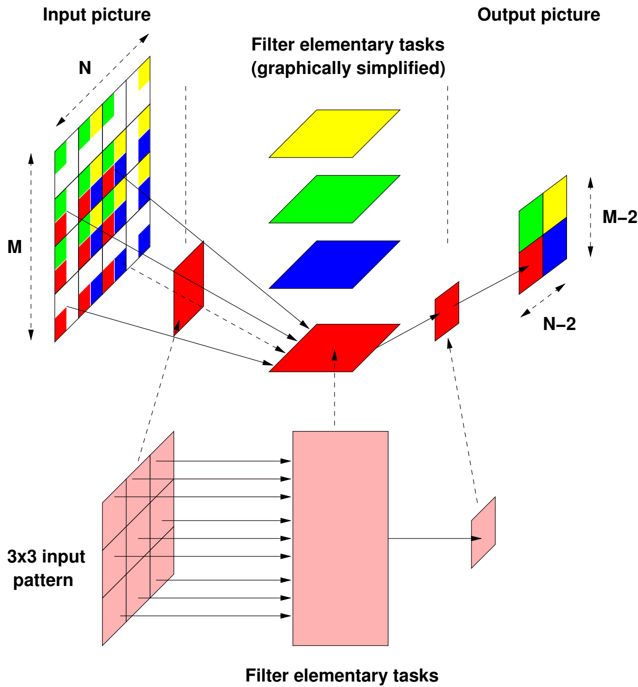

Fig. 1. Image filtering application. The top of the figure illustrates a repetitive iteration for the image filtering application. This application consumes an M×N input image, on the left hand side, and produces an (M-2)×(N-2) image, on the right hand side. Parallel iterations of the task are illustrated in the top center. The color of each task iteration identifies the corresponding data consumed and produced in the input and output image. The bottom side of the figure represents one task, a 3×3 consumed *pattern* (read in the input image) and the output *pattern* composed of a single pixel, which is written in the output image. Both consumed and produced *patterns* result from the application of a *tiler*.

III-B. Motivating Example: Image Filtering

We illustrate the ARRAY-OL application design in Gaspard using the image filtering academic example. The objective is to correlate well known filters (Gauss, Croix, Sobel, Prewitt, etc.) with images. This example is interesting and relevant because of the multi-dimensional arrays and computations managed: the execution of a spacial two dimensional filter and the temporal dimension of the repetition above the image. Therefore, three data and computation dimensions are managed.

Figure 1 details this application in ARRAY-OL. The top of the figure illustrates the repetitive iteration of a parallel task, while the bottom side illustrates one iteration at the *data parallelism* level. At this level, the *patterns* (multi-dimensional data consumed and produced by the elementary tasks) appear. These *patterns* are built according to the *tilers*. As opposed to usual sequential loops, the repetition specification does not induce any artificial scheduling for task execution and data transaction.

III-C. Gaspard Targets FPGAs

Several processor based execution models have been introduced in Gaspard to compile an ARRAY-OL application, allowing execution on processors. With the introduction of FPGA in the Gaspard framework, it becomes necessary to study the feasibility for an ARRAY-OL hardware and dedicated execution model. Especially, we have to manage the multi-dimensional arrays and repetitions specific to ARRAY-OL, to efficiently transform this model into hardware, and to define a VHDL code generation from this hardware model.

Our flow performs efficient transformations, FOLDING and UNFOLDING, that modify loops and repetitive structures in order to optimize an hardware implementation according to an FPGA. Transformation does not introduce control in the datapath computing resource. Control is introduced in the interface with the sensors. Therefore, the static task scheduling is still expressed in the ARRAY-OL language, keeping all the parallelism expression provided by this language.

IV. HARDWARE EXECUTION

The ARRAY-OL application modeling detailed in the previous section is not directly executable on an FPGA since the repetition expression contains the temporal dimension, furthermore the target FPGA characteristics are not considered. We present a flow that transforms the initial ARRAY-OL application into hardware, allowing an FPGA implementation. By introducing the FPGA characteristics in the flow, we enhance the application hardware implementation. The flow is illustrated on the Figure 2 and is composed of the four following steps:

- the **connection** step analyzes the data dependencies expressed with the *tilers* and generates the appropriate hardware connections;
- the **computation** step adjusts the application implementation with the FPGA, and maximizes its computation power;
- the **routing resources** step adds routing elements on IO when necessary;
- the **code generation** steps generates the VHDL code, which is synthesizable onto a FPGA.

IV-A. Connection Step

In ARRAY-OL, exact data dependencies are expressed by the *tilers*, providing synchronizations between tasks. Technically, *tilers* define the way data are read into an array to produce a *pattern*, through *origin*, *paving* and

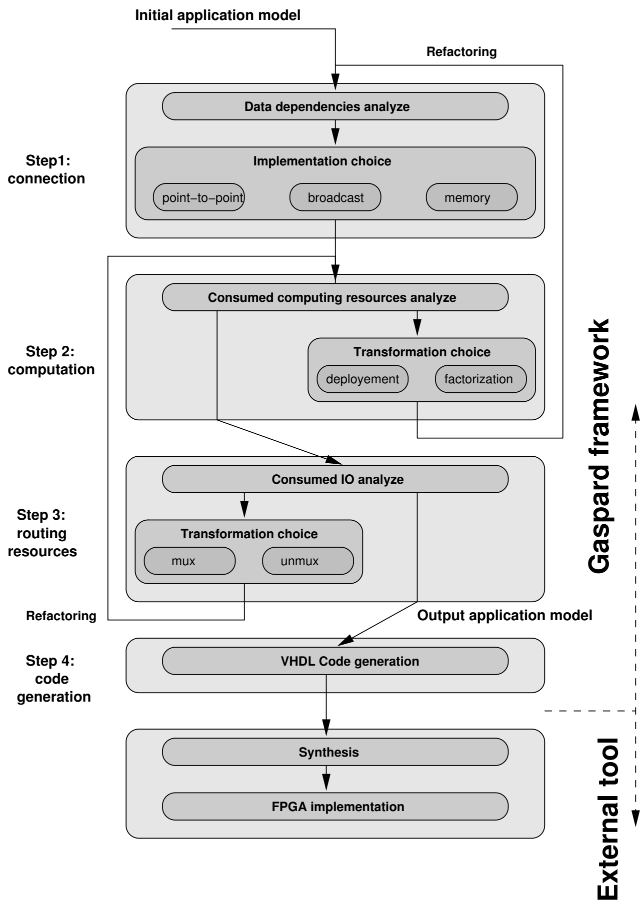

Fig. 2. Our automatic design space exploration flow. It is integrated in the Gaspard framework and is composed of four steps. Its aim is to transform the initial application model into hardware, to optimize this hardware implementation and to generate the corresponding VHDL code. From the generated code, we use external tools to realize the RTL simulation, the synthesis effort and the FPGA implementation.

fitting matrix. We refer the reader to [14] for precision on the analytical formulation. For a dedicated hardware execution, that does not support dynamic data transaction, *tilers* are compiled before the execution in order to create a static connections topology. The aim of this step flow is to provide such hardware connection topology for data dependencies expressed in the initial application.

In the connection flow, the data flow described in ARRAY-OL with *tilers* is transformed into hardware elements and blocks. These hardware elements provide a precise quantification of the hardware resources required for an implementation. However, the resulting implementation does not take into account the FPGA characteristics. The computation step transforms the application according to a particular FPGA.

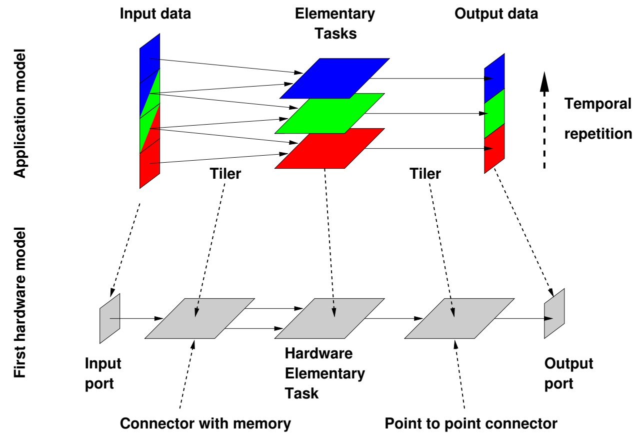

Fig. 3. From the application input model to its hardware implementation using *tiler* connector. According to the top left-hand side of the figure, data are read two times on the temporal dimension. Thus, a “connector with memory” is necessary, and a shift register is automatically created. Looking at the output, we observe that each produced data is written only one time, allowing us to not store the data and to use “point to point connector”. The temporal dimension around the task is compiled as a single hardware task.

IV-B. Computation Step

The computation step evaluates the computing resources required for a particular implementation. If this evaluation is not compatible with hardware resources, a transformation (refactoring) of the application model is required. Both synthesis and estimation provide the computing resources required for an implementation. In order to perform a fast design space exploration, we use estimation results proposed in [22] for each elementary task contained in the application model. This estimator provides an atomic estimation, not a global one, since transformed tilers and the repetition mechanism, which introduces multiple instances of one task, are not managed. Exploring the application model, we perform this global estimation and compare this result with the target FPGA characteristics. Therefore, we obtain a percentage of FPGA area required for an implementation. Two transformations allow to “refactor” the application at the model level. The UNFOLDING transformation increases the implementation size, while the FOLDING one reduces it. Therefore, it is feasible to fit the number of used resources in order to implement the most powerful executable solution on a particular FPGA. Figure 4 illustrates the FOLDING and the UNFOLDING transformations.

ARRAY-OL transformations and estimation tools have

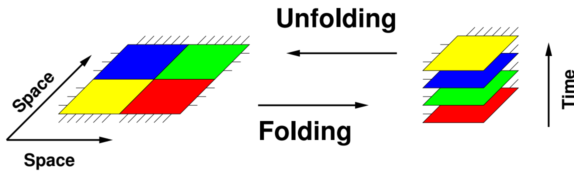

Fig. 4. Transformation that enables application refactoring. The UNFOLDING transformation increases the number of computing resources exploited for a hardware implementation, while the FOLDING one reduces this number.

been developed in the team, and we are currently working on linking our design flow with those tools. We refer the reader to [23] and [14] for the ARRAY-OL transformation description and the way they may be driven from the Gaspard environment.

The computation step of the flow compares the area required for an hardware implementation with the one offered by the target FPGA. This step performs application refactoring if necessary. However, it does not compare the inputs and outputs ports required by the implementation with the number of ports provided by the FPGA.

IV-C. Routing Resources Step

Recent FPGAs contain several hundred of inputs and outputs (IOs) ports dedicated to user. For most part of the application, there are enough IOs. In systematic signal processing applications, where data can be produced by multiple identical sensors, like in multi-dimensional detection applications, the numbers of IO is a restrictive parameter. A way to consume less IOs is to multiplex the data streams before entering data into FPGA. Inside FPGA, the data streams are de-multiplexed, offering possibilities to realize all computation in a parallel way. This data multiplexing/de-multiplexing is called routing elements in this study, and is illustrated in the Figure 5.

According to the modification introduced by the routing resources, the consumed computing resources is necessarily modified. The design flow then iterates to the second step to take this modification into account (Figure 2).

IV-D. Code Generation Step

When the third step is successfully completed, the VHDL code generation is launched. The generated code allows to simulate and synthesize the resulting application. With a synthesis tool, an FPGA can be configured. This code generation has been successfully validated with several relevant applications inspired from industrial situations, such as the correlation algorithm detailed in [22], matrix multiplications, and the image filtering application taken as an illustrative example in this paper. The automatically generated VHDL code keeps the parallelism

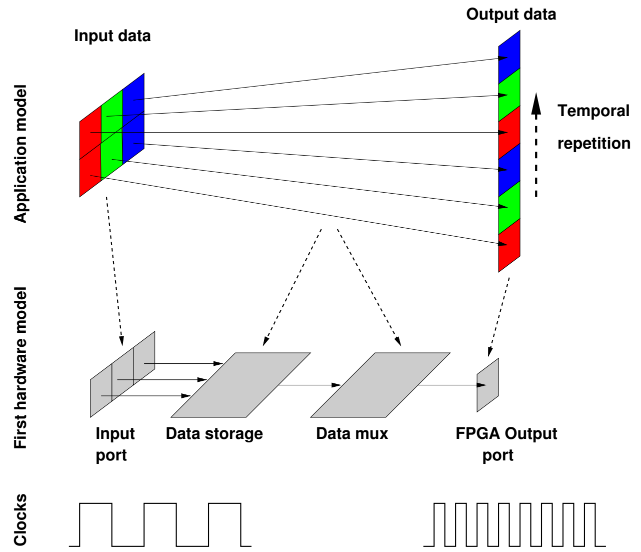

Fig. 5. Data multiplexing toward routing resources. The top of the figure illustrates the data that are read on the spacial and temporal dimensions, and written on the only temporal dimension. For each data input clock cycle, three data are produced on the spacial dimension. They are stored and cyclically sent to the output through a multiplexer. The data output clock cycle is three times higher than the data input one.

expression provided at a high level description. Moreover, multidimensional arrays manipulation are managed.

In this section, we have shown the different steps necessary for an efficient implementation of an input application model and the targeted FPGA. We have identified and illustrated refactoring transformation. From a VHDL code generation, we are able to synthesize and implement the resulting hardware design onto an FPGA. The next section describes a usage of the flow on the application example introduced in Section III-B.

V. FPGA MULTIPLE ABSTRACT VIEWS

Our design space exploration flow automatically produces an hardware design according to an initial application model. The first step directly produces a hardware design, with no FPGA characteristics consideration. The second and third steps compare the computing resources and the IOs required for an hardware implementation with respect to the resources offered by the FPGA. According to these results, application refactoring are triggered, providing the most efficient hardware implementation that fits onto the target FPGA. For this purpose, two different views of an FPGA are introduced: a BLACK BOX and a QUANTITATIVE view. In the following, we detail these two views and introduce a third one, the PHYSICAL view, that allows

to perform a mapping of an application onto an FPGA according to the FPGA geographical details.

V-A. Black Box View

The BLACK BOX view provides the simplest view of an FPGA. While implementing a design onto FPGA, no regards to the FPGA details are considered. This view allows to model complex and heterogeneous architecture that includes reconfigurable component [24], provides a fast and direct mapping of a design and allows to manipulate FPGAs with unknown characteristics. However, while some FPGAs are powerful enough to implement a given design, some others could fail during the synthesis effort, introducing an important lack of time. By the same way, a given design could not be implemented by the most efficient way. The BLACK BOX view is used in the first step of our flow, Figure 2 and Section IV-A.

V-B. Quantitative View

The QUANTITATIVE view provides a list of each resource included in an FPGA. Each kind of resource (i,e RAM-512, RAM-1k, DSP blocks, IOs), that composes an FPGA cell appears in this view. However, configurable wires are not considered because of their too high complexity, heterogeneity and density. Not considering such wires may introduce a failure during synthesis effort because of the potentially high connection congestion. However, current FPGAs provide enough configurable wires to support high density connection design. The QUANTITATIVE view makes possible the application tuning, by performing an implementation that corresponds to the available FPGA resources. The QUANTITATIVE view is used in the 2nd and 3rd steps of our flow in order to perform refactoring, Figure 2 and Sections IV-B and IV-C.

V-C. Physical View

The PHYSICAL view is the most precise view manipulated in this study. BLACK BOX and QUANTITATIVE views are constructed from the PHYSICAL one.

To create a PHYSICAL view, an FPGA is considered like a two dimensional grid of cells. Each cell functionality is known (like in the QUANTITATIVE view), and is linked to its position in the grid. With the whole individual cells, a very precise view of an FPGA topology is constructed. Current heterogeneous FPGA architectures can be modeled, and mapping heuristics described in [15], [17], [16] can be used. According to the data dependencies constraints, the PHYSICAL view makes possible the design mapping onto an FPGA. The aim of this mapping, constructed from high level evaluation of data dependencies, is to provide a more efficient placement than the one provided by the FPGA mapping tool, which does not consider data dependencies issued from hierarchy, like demonstrated by Moeller in [11]. For this purpose, we integrate in our flow a mapping constraint file that guide the mapping issued by the synthesis tools.

Our flow is modified in order to consider the PHYSICAL view. The 4th step now includes two sub-steps: the remaining VHDL code generation and the file constraint code generation. While the generated VHDL code can be used in either synthesis tool, the constraint file does not. To constraint a mapping, the component instances and the number of required specific cells and the position in the grid are manipulated. Manipulating the component instances in the VHDL generated code is possible thanks to the hierarchy kept from the initial application. The mapping of a design onto an FPGA physical view provides an intermediate virtual view, very similar to Lagadec *et. al.* [25]. We gather the FPGA cell to provide a virtual architecture that match the application grain. Therefore, each module contained in the initial application that is clearly identified in the VHDL code is also now clearly identified on the FPGA as a gathering of cells. The generated placement constraint file is provided to the synthesis tool with the VHDL code. Figure 6 illustrates our complete flow.

Our approach is illustrated with an application that shares input data. These inputs data are sent to two non-dependent computation units, that realize the correlation from the Eq. 1 and 2. While providing to the synthesis tool the single VHDL code generated, we observe that the application is mapped in the center of the FPGA, with no regards to the shared data and the module decomposition that appears in the VHDL code, Figure 7. The unconstrained solution does not provide a structural implementation, disabling the opportunity to exchange a module functionality. Using a file constraint, we gather shared data in high density FPGA columns, with the computation units around them. The structure contained in the VHDL code perfectly matches the structure expressed for this constrained implementation. The synthesis results have demonstrated the equivalent performance for both implementations, with a decrease of the area consumed by the constrained one. This area consumption decrease is linked to the high density implementation required by the constraint mapping. The main advantage of the constraint implementation is the module decomposition that provides an opportunity to enhance in some cases the implementation efficiency, to modify the module functionality and, therefore, to introduce partial reconfiguration in our flow.

$$C_{cy}(j) = \frac{1}{N} \sum_{i=0}^{N-1} c(i) \cdot y(i+j) \tag{1}$$

$$C_{dy}(j) = \frac{1}{N} \sum_{i=0}^{N-1} d(i) \cdot y(i+j) \tag{2}$$

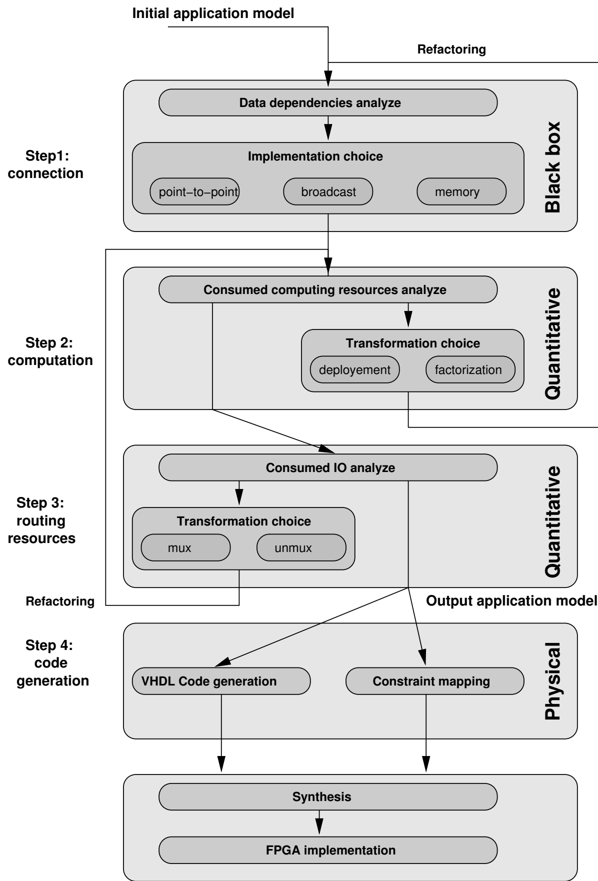

Fig. 6. Our extended automatic design space exploration flow with the corresponding BLACK BOX, QUANTITATIVE and PHYSICAL views. Similarly to the VHDL code generation, a file that contain mapping information is generated. This file provides constraints to the synthesis tool on the design placement. These constraints are issued from a high level evaluation of the parallelism included in the application and the topology provided by an FPGA.

This automatic module creation can also benefit from the regularity expressed at a high level model (using ARRAY-OL) and kept until the VHDL code. Especially, a regular nested loop can be very efficiently implemented onto an FPGA that also contains regularity. The next contribution of our flow is to match the regularity contained in an application with the one contained in an FPGA. Moreover, the incremental synthesis option, that re-use component synthesis results for its instances in order to considerably reduce synthesis time, will be performed.

The PHYSICAL view allows to identify the functionality of each FPGA cell and its associated configuration at a given mapping. Until now, Gaspard applications are expressed using the ARRAY-OL model, which is purely data-flow. Using an extension that introduces a control flow

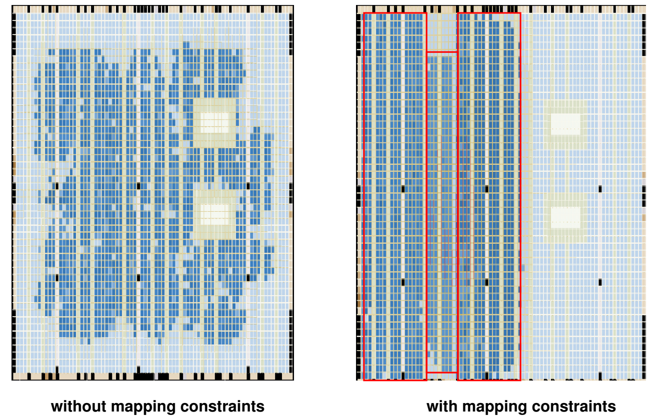

Fig. 7. Synthesis results with and without mapping constraints. Without constraints, the synthesis tool provides a design placed in the middle of the FPGA, with no regards to the application data dependencies. Using constraints, we are able to gather shared data in a single high density FPGA column with the computations units around them.

in the data flow [26], [27], we identify, in a design, the activity of an application (i,e active, inactive). Therefore, according to the PHYSICAL view and the control flow, we believe that it is possible to dynamically and partially reconfigure an FPGA.

VI. CONCLUSION

This paper presents an approach to manipulate reconfigurable resources such as FPGAs in the Gaspard codesign framework. The design space exploration flow is composed of four steps. The 1st one identifies data dependencies in the application specification. The 2nd and 3rd steps compare the computing resources and the IOs required for a hardware implementation with respect to the resources offered by the FPGA. According to these results, application refactoring are triggered, providing the most efficient hardware implementation that fits onto the target FPGA. The last step generates the VHDL code.

In this paper, we have extended our flow in order to perform an FPGA mapping of parallel application modeled at a high level. For this purpose, we have introduced three FPGA views that allow to manipulate an FPGA according to the designer requirements: the BLACK BOX view for a first rought design, the QUANTITATIVE view for a tuned hardware design and the PHYSICAL view for an FPGA mapping constraints. The PHYSICAL view is the most detailed view considered in this study. It allows to map an application onto an FPGA in a modular way, taking into account data dependencies and application structure. We believe that FPGA implementation issued from such mapping constraint could benefit from high level applica-

tion modeling and data dependencies expression. Moreover, this mapping module composition may be used to introduce the partial reconfiguration and a regular mapping.

The future works will provide a fully automatic flow and will extend the flow to the partial reconfiguration and the regular mapping. The introduction of a control flow into our data flow language should allow to perform partial and/or dynamic reconfigurations.

VII. References

[1] T. Beierlein, D. Fröumlhlich, and B. Steinbach, "Model-driven compilation of UML-models for reconfigurable architectures," in *2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04)*, Toronto, Ontario, Canada, May 2004.

[2] WEST Team LIFL, Lille, France, "Graphical array specification for parallel and distributed computing (GASPARD-2)," http://www.lifl.fr/west/gaspard/, 2005.

[3] A. Dias, C. Lavarenne, M. Akil, and Y. Sorel, "Optimized implementation of real-time image processing algorithms on field programmable gate arrays," in *4'th Int. Conf. on Signal Processing, ICSP'98*, Beijing, China, 1998.

[4] L. Kaouane, M. Akil, Y. Sorel, and T. Grandpierre, "From algorithm graph specification to automatic synthesis of FPGA circuit: a seamless flow of graph transformations," in *13th international conference on Field-Programmable Logic and Applications, FPL'03*, Lisbon, Portugal, Sept. 2003.

[5] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, Sept. 2004, pp. 7–16.

[6] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, and D. Stroobandt, "From loop transformation to hardware generation," in *Proceedings of the 17th ProRISC Workshop*, Veldhoven, Nov. 2006, pp. 249–255.

[7] A.-C. Guillou, P. Quinton, and T. Risset, "Hardware synthesis for multi-dimensional time," in *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors (ASAP 03)*, The Hague, The Netherlands, June 2003, pp. 40–51.

[8] Project Inria-CNRS COSI, "ALPHA home page," http://www.irisa.fr/cosi/ALPHA/.

[9] K. R. Shesha Shayee, J. Park, and P. C. Diniz, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, Napa, CA, Apr. 2003, p. 296.

[10] J. Park, P. C. Diniz, and K. R. Shesha Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1420–1435, 2004.

[11] T. J.Moeller, "Field programmable gate arrays for radar front-end digital signal processing," Ph.D. dissertation, Massachusetts Institute of Technology, May 1999.

[12] B. So, M. Hall, and P. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

[13] H. Ziegler and M. Hall, "Evaluating heuristics in automatically mapping multi-loop applications to FPGAs," in *ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA'05)*, 2005, pp. 184–195.

[14] P. Boulet, "Array-OL revisited, multidimensional intensive signal processing specification," INRIA, Research Report RR-6113, Feb. 2007. [Online]. Available: http://hal.inria.fr/inria-00128840/en/

[15] M. Handa and R. Vemuri, "A fast algorithm for finding maximal empty rectangles for dynamic FPGA placement," in *Design, Automation and Test in Europe Conference and Exhibition, DATE'04*, Paris, France, Feb. 2004, pp. 744–745, Vol.1.

[16] J. Tabero, J. Septién, H. Mecha, and D. Mozos, "Task placement heuristic based on 3D-adjacency and look-ahead in reconfigurable systems," in *11th Asia and South Pacific Design Automation Conference, ASP-DAC 2006*, Yokohama, Japan, Jan. 2006, pp. 384–389. [Online]. Available: http://doi.acm.org/10.1145/1118299.1118397

[17] R. Tessier, "Fast placement approaches for FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 2, pp. 284–305, 2002.

[18] P. Dumont and P. Boulet, "Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II," INRIA, Research Report RR-5516, Mar. 2005. [Online]. Available: http://www.inria.fr/rrrt/rr-5516.html

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, LUSTRE*: a declarative language for real-time programming*. ACM Press, 1987.

[20] P. Le Guernic, J. Talpin, and J. Le Lann, "Polychrony for system design," *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, Apr. 2003.

[21] INRIA, "DaRT short presentations and demos," http://www.lifl.fr/west/DaRTShortPresentations/, 2007.

[22] S. Le Beux, V. Gagne, E. Aboulhamid, P. Marquet, and J.-L. Dekeyser, "Hardware/software exploration for an anti-collision radar system," in *The 49th IEEE International Midwest Symposium on Circuits and Systems*, San Juan, Puerto Rico, Aug. 2006.

[23] P. Dumont, "Spécification multidimensionnelle pour le traitement du signal systématique," Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, Dec. 2005.

[24] MORPHEUS project, "Multi-purpose dynamically reconfigurable platform for intensive heterogeneous processing," http://www.morpheus-ist.org/.

[25] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier, "Placing, routing, and editing virtual FPGAs," in *11th International Conference on Field-Programmable Logic and Applications, FPL'01*, Belfast, Northern Ireland, UK, Aug. 2001, pp. 357–366. [Online]. Available: http://www.springerlink.com/content/rvx89nt37g7ln6wr/

[26] O. Labbani, J.-L. Dekeyser, P. Boulet, and E. Rutten, "UML2 profile for modeling controlled data parallel applications," in *FDL'06: Forum on Specification and Design Languages*, Darmstadt, Germany, Sept. 2006.

[27] A. Gamatié, E. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser, "Synchronous modeling of data intensive applications," INRIA, Research Report 5876, Apr. 2006.