Intermediate level components for reconfigurable platforms

Erwan Fabiani, Christophe Gouyen and Bernard Pottier
Université de Bretagne Occidentale, Brest, France.
fabiani@univ-brest.fr, gouyen@univ-brest.fr, pottier@univ-brest.fr

***Abstract*— This paper addresses the problem of development productivity on reconfigurable platforms. Due to the availability of generic low level tools and powerful logic synthesis tools, it becomes possible to define portable components that have both a high level behavior and attributes for physical synthesis.**

The behavior of a component can be fixed at compile time using concise specifications that will reduce the cost and delays in developments.

The method allowing to produce components is illustrated with two case studies.

I. Introduction

We are considering a new generation of general purpose circuits allowing to produce applications by field programming or configuration. Following FPGAs, the economic challenge of these circuits is to complement ASIC for markets where the production volume does not balance the cost of a specific SOC design, and where a quick application availability is critical. A consequence of cost and time-to-market constraints is the need to define software production methods with emphasis on designers productivity.

A. Scope

Among the different architectural options appearing, or likely to appear, we are selecting a general framework with the following parts (figure 1) :

1) a dedicated system processor (SP) in charge of tasks and circuit management,
2) a network on chip possibly simple and controlled by SP,
3) several heterogeneous compute units (CU) such as processors, reconfigurable data path or fine grain banks. These units have their own local memories for data, and code or configurations.
4) a memory cache ,
5) several input/output units with, possibly, specific support outside the circuit.

There are two main motivations for the choice of such a distributed architecture.

One is scalability, with the need to have an evolving choice of off-the-shelf circuits adapted to different kind of applications. A permanent problem with current FPGA technology is the change of scale and the actual difficulty to implement system level communications in an efficient way[5]. The use of a network on chip[1] allows to merge SOC IPs within the platform, and to avoid congestion in the routing resources during system activities.

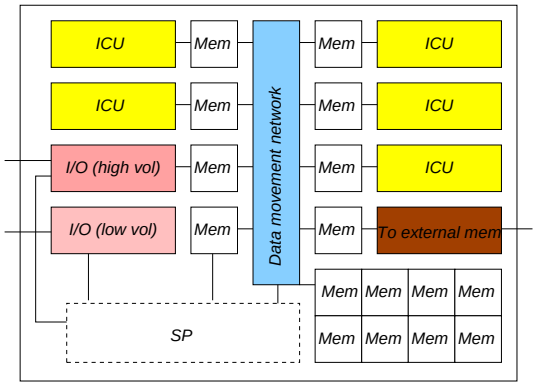

Fig. 1. A parallel heterogeneous reconfigurable platform as proposed by O. Villellas

The other motivation is heterogeneity, meaning that it will be possible to select a set of reconfigurable or programmable resources suiting the application. This implies the development of tools allowing to produce code or configurations from a single specification, depending on the architecture and execution constraints.

B. System behavior

The platform must support compute intensive processes as found in stream, image and signal processing. These processes will use on chip input/output facilities, buffers, memory buffers, and they can spread over several compute units. In this case they need to be prepared as small tasks exchanging data buffers or transactions. Intensive computation tasks will be mapped to reconfigurable units. A Kahn process decomposition is an adequate approach for preparing such a program structure[7].

Other processes needing specific hardware support are controllers having short reaction delays.

The operating system decides resource allocation, scheduling, swaps and memory transfers.

C. Code generation for units

During these last years, our research activity has been concentrated on building portable tools for reconfigurable architectures. This activity is part of different projects but we retain the concepts and developments in a long term framework called Madeo.

Madeo is organized in three parts (see figure 2)..

1) The lower layer proposes tools for reconfigurable architecture modeling. Several fine grain FPGA architectures have been described successfully, including commercial circuits. The models are represented using a grammar that enables a set of generic tools to produce the basic functionalities : placing cells on FPGAs, global or point to point routing, floor-planning, regular circuit design[10]. An important property of this framework is its openness allowing synthesis algorithms to build layout of application component under programmer control.
   Fine control on the geometry and location of components is critical for the design of resource management, as it is needed in operating systems.
2) Above these tools, there is the support for logic synthesis. The second layer uses high level object oriented specifications and produces hierarchical application components for the first layer. The basic flow is based on directed acyclic graph of nodes representing procedure calls that will be translated into look-up tables (LUT) or call of other graphs.
   The second layer tools can work competitively compared to handwritten hardware implementation because data specifications are required to be richer than usual types. Our data types are based on set of values and intervals. They are automatically produced for each function in the program and propagated downward the hierarchical graph.
   After type inference, synthesis tools have an exact knowledge of the computation context, and are able to lead very efficient optimizations. These optimizations take place at a symbolic level, by collapsing and simplifying nodes in the LUT graph, and at the encoding level, by exchanging data for indexes, and finally at the logic level using logic synthesis algorithms[4] such as those packed in SIS[19].
   Translation schemes for fine grain FPGAs have been described in [11]. Extension for reconfigurable data path code production is currently being investigated from the same set of tools as the type system also provide support for interval description.
3) Above these levels we are now interested to develop architecture *components* in different ways. This paper will discuss the component status, especially their important position in the design flow and the relation with the physical target (possibly, CU from the platform).

D. Component definition

Component are intermediate in the application design flow. They can be used to articulate the migration from software to hardware in a transparent way for application developers. Their main characteristics are described as follow.

a) **Modularity and reuse:** Components provide a modular behavioral interface usable during application development, either directly or from a compiler. They are defined as objects grouping a behavioral interface, physical synthesis capabilities, rules for use, and code or configuration to be handled by the operating system.

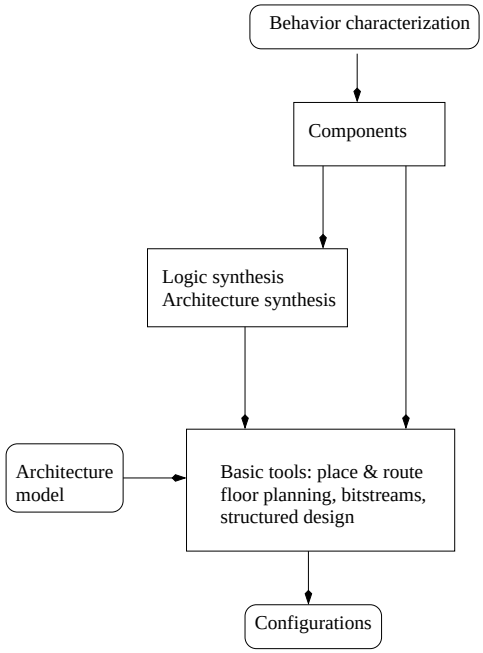

Fig. 2. Position of the component layer related to synthesis tools and basic tools

Components provide *software re-usability*, in a way similar as IP modules do in the case of SOC.They are executable in the software development environment.

b) **Programmability and characterization:** The component behavior can be defined at compile-time (or run-time) using domain specific languages (DSL). Alternatively, there can be a fixed parametrized behavior. In each case, components carry an implicit execution architecture that will be produced at the physical level.

Software macros as used in the FPGAs environments are components of small complexity whose definitions are hidden to the programmers.

c) **Physical synthesis:** Components embed algorithms producing a physical description of the application architecture related to a reconfigurable unit target.

These algorithms use building blocks in the form of other components, or specific placed and routed primitives. They compute the respective layout of these blocks, and they produce the low level interconnections.

Physical synthesis algorithms are portable at least for fine grain architectures.

d) **Support for compilers:** Some components are explicit structured descriptions of hardware. It is the case for arithmetic operators, regular processing networks, controllers.

There are also components representing the necessary transformations enabling a compiler to produce circuits in a restricted context (computation graph, regular networks to be mapped on Us).

II. Productivity is concise specifications

As development productivity is becoming a serious challenge for embedded applications, it is interesting to observe how software has solved this difficulty by the past, then in which way reconfigurable architectures could help in speeding up the development process.

A. Productivity in software development

An important factor in development productivity is the level of abstraction in which solution specifications are produced. The first benefit of abstraction is the simplicity of the expression obtained due to the meanings of the formalism. Simplicity means speed and security of solution expression, ease to develop and maintain translation and verification tools. Programming languages have achieved gradual progress in terms of abstraction level and in terms of modularity and reuse.

Software productivity can be evaluated on metrics such as *line of codes*, or may be more accurately on *source statements* implementing equivalent functionalities. According to industrial expert sources, productivity can scale from one to ten for general purpose languages, and it is not necessary to insist on the power of expression of specific languages.

There exists at least two clear demonstrations of the interest of abstraction rising given by *virtual machines* for general purpose languages, and *domain specific languages*. In either cases the basic support is provided by of a particular virtual architecture or software supports that provides a fixed higher semantic level (see figure 3). The compiler design is usually simple due to the service offered by the underlying support and the language can be ported to different platform by adapting this support.

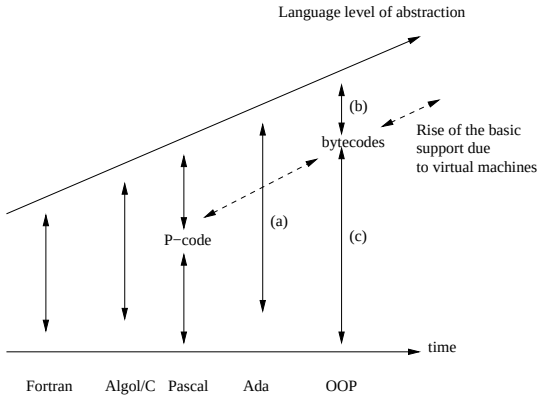

Fig. 3. Compiler complexity in the cases of direct translation and virtual machines. (a) filling the semantic gap is the compiler responsibility. (c) a virtual machine rises artificially the level of basic support. (b) the compiler tasks is considerably reduced, porting is easy and the code is small.

e) Virtual machines: can be implemented as pure software, as it has been the case for languages as Pascal (P-Code), Modula-2, Smalltalk-80 (byte-codes), Java, Lisp. Alternatively the pseudo code can be implemented efficiently in processor micro-code as it was the case for Smalltalk on the Xerox PARC Dorado[12], or some Pascal and Java implementations. Occam hardware support on the Transputer processors also relied on a micro-coded instruction set supporting the paradigm of processes and channel communications[15].

Virtual machines group together an interpreter (or micro-code) for the target language, memory and machine support, and primitives. Primitives are designed to bring up new functionalities to be addressed from the instruction set, or to accelerate computations related to the interpreter performance. Primitives can also rely on hardware supports.

f) Domain specific languages : (DSL) are largely used in our current software tools, producing the desirable level of abstraction related to a particular domain[20]. Examples of DSL are text processing tools (sed, awk, etc...), compilation tools (lex, yacc, ...), or more specific domain tools for signal processing, graphics, etc...Due to their capability to change the application architectures, DSL can be implemented on reconfigurable platforms.

Drawbacks in using DSL include the excessive specialization of data. Testing can be an issue if the formalism does not support associated execution mechanisms, and finally DSL abstraction does not avoid domain expertise from their programmers. It just ease the task.

III. Concision on reconfigurable platforms

The productivity objective leads to anchor the application development process in the software world rather than in too much general hardware description. DSL and high level languages have been used in the CAD tools practically from the earliest days of VLSI[13]. However the absence of open tools for reconfigurable architecture has discouraged their development leaving very few public software[2].

To come back to our platform, let us envision the places where specific supports could be used with profit in application developments :

- The operating system (OS) is in charge of the circuit management, that involves feeding units with configurations, code and data, controlling the network, cache memory handling. This involves a lot of specific operations some of them related with hardware control. The other attribute of the OS is the process management. Each process uses memory and computing resources, makes use of i/os operations and synchronizations,
  Synchronizations and communications, resources management encourage the use of object oriented and process oriented (events) specification. Circuit and execution control is also a good place for a virtual machine handling the hardware from specific instructions.
- Stream computations are an important activity for this kind of platform[7]. Streams are data flow coming from network or sensors on which processing is applied with few side effects.
  The functional style of programming is generally far more productive than the imperative one, and appears to be adequate for data flow computations.
- Signal processing algorithms can be handled by direct

translation of the mathematical specification into computation graphs for fine or coarse grain reconfigurable units.
– Grammar oriented tools has yet been successfully used to specify and synthesize communication protocols[16], [17].
  These tools define both the format and nature of data exchanged during the communications, and the automata handling these communications. They can be useful either at the application level for handling packets assembly, and at the system level to enable transactions and communications on the platform.
– Pattern matching, image processing,
– Cellular automata is a simple data parallel model that describes massive computations from a neighborhood description and transition functions.
  They support higher level algorithms (physic modeling, image processing). See section IV-B.
– Systolic and data parallel languages local behavior and exchanges.
– Sequential behavior, such as loops operating on a restricted set of variables.

One could envision the software environment with Un*x, or object oriented glasses. In Un*xes, tool composition is based on pipes connecting text streams. This cannot be efficiently reproduced in the run-time environment of an embedded system since the cost of conversions from binary format to texts is high and provides few interests.

Decoding and encoding data on the fly can be handled automatically if a formal specification is produced for these data. This is a solved problem thanks to communication tools based on of ASN.1 for binary presentation, Express for data specification[18], or CORBA object brokers.

IV. Component design method

The general approach for component design is bottom-up :

1) Fix the functionality to be addressed by defining what will be explicit in the parameters or program, and what will be implicit.
2) define the internal execution model
3) define the language
4) define the synthesis mechanisms related to a support architecture.

To help the explanations, this approach will be illustrated on the example of cellular automata on the ArMen computer.

A. Platform description

In this case, the computer is ArMen, a distributed memory architecture whose nodes are fully interconnected using serial links (figure 4). Each node processor has an attached FPGA accessed in locked step read or write transactions. The processor has also support in its address space to write and read configuration to the FPGA (figure 5).

FPGAs are connected together and can exchange data asynchronously, there is no global clock. The interface from the local system bus to the FPGA is fixed and generally used to feed a pipeline. Inside the FPGA, physical synthesis generally

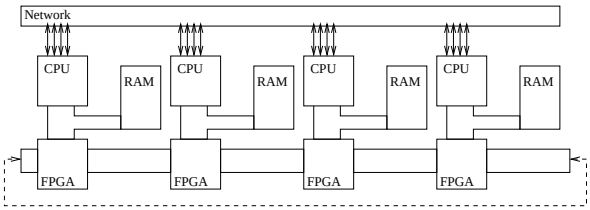

Fig. 4. Four interconnected nodes

proceeds by allocating logic resources along a pipeline using local routes. The pipeline stages are connected to long lines by three state buffers. Long lines is an internal bus that can bring back results to the interface with short delays.

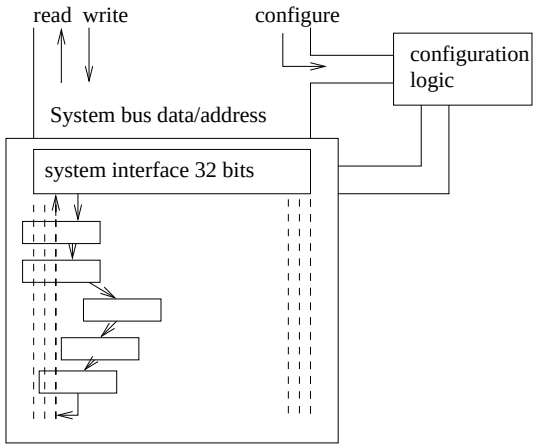

Fig. 5. Physical representation of pipelines inside the FPGA

The vertical pipeline advances under processor control while other computation usually take place horizontally in an asynchronous way[6].

B. Functionality

Cellular automata (CA) is a well known paradigm where a cell space progresses synchronously, step by step. To define a CA it is needed to fix :

1) a neighborhood representing the dependencies relative to a cell,
2) a transition function describing the evolution of a cell given its current state and the neighbor states.
3) the geometry of the cell space and its initial value.

A CA specification must *explicit* these three points letting the component implement a massive parallel computation or alternatively observe where new computations are really needed and achieve these computations.

C. Execution model

We use the massive parallel model with a locally parallel, globally sequential approach. The data space is divided in stripes recorded in node memories. The width of the stripes is the bandwidth to the FPGAs (32 bits $\times$ number of nodes). One or two nodes are in charge of feeding data dependencies

on the slice borders and to read back these dependencies for the future step.

Processors manage two spaces for current state and next state. Their activity is to repetitively transform their current stripe into a new one. They need to exchange values because of the dependencies on the stripe borders.

D. Program expression

Programs are expressed in a simple syntax covering the 3 definitions given section IV-B :

1) each cell state is described as a C record grouping bit-field variables.
2) the neighborhood is declared as a set of directions (C, for center, N for north, NW for north-west…),
3) the data space is declared by two integer values for with and height,
4) the transition function is a C function returning the new value of the local cell computed from the neighborhood state.

E. Architecture description

As CA can be considered as fine grain computations involving a lot of data exchanges, the transition function will be synthesized in hardware. To enable this function to proceed, it is needed to present the neighborhood. Thus, a simple approach for the architecture is to provide a FIFO in which the cells are progressing, and to connect this FIFO to a row of processors. Dependencies can be wired between adjacent nodes.

Control is the responsibility of the processors that permanently read their memories, write to the FPGAs, read back the new state from the FPGA to write it to memory. Their coordination is enforced during the accesses by local handshakes. Several time steps can be cascaded along a pipeline.

Interested reader can find more details in [3].

F. Physical synthesis

Architecture implementation is a fully automatic process leaded by a dedicated synthesizer. The availability of tools described in section I allows a constructive approach of the physical synthesis. The constraints that need to be observed are the cell width, the data path width in the FPGA, the size of the processors, the possible saturation of routing resources.

Physical layout can be achieved following these steps :

1) synthesize the processor,
2) place and route the processor,
3) compute the FIFO size,
4) make an estimation of the routing channel width,
5) place the processors on the FPGA dye, place the FIFO registers,
6) call the point to point router to connect registers together, connect the registers to the processor, connect the processor to the feed back lines, connect the registers to the interface.
7) connect the clock to the interface

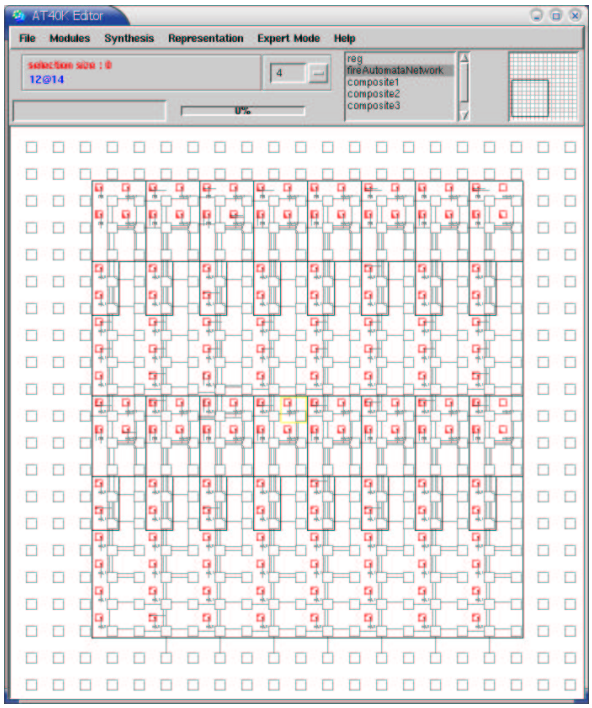

Fig. 6. Layout of a cellular automata for fire propagation on an Atmel 40K FPGA. The circuit has small processors which internal state represent the situation on 2 bits at a geographic position. Two stages of automata had been cascaded, and 8 slices are represented. The same physical layout code is usable on other FPGAs with quite different results. The AT40k cells group two 3-bit LUTs. This view is produced from the current Madeo low level tools and there is no attempt to achieve a connection with i/o pins or an interface.

G. Stacking components

Models can be stacked. As an example, we have produced a partial implementation of the Wu and Manber pattern matching algorithm[21] above the CA component. In this case, the program becomes a pattern to be searched, and the number of errors that are accepted.

Implementation of some low level operators for image processing is also immediate.

H. Physical and computational constraints

Physical synthesis for a component is a determinist approach dealing on one side with the reconfigurable unit organization and resources, and on the other side with a characterization of the component. The behavior of the component is fixed by a high level program block processed by the logic synthesizer.

Another important issue is the internal development of parallelism during synthesis. Care must be taken not to waste hardware resources by adapting synthesis algorithms to the usable data rate on the unit interface.

V. Physical synthesis for Systolic arrays

A. Example for systolic arrays

This part shows for a specific applicative model (systolic arrays) what could be the advantages of having specific

component definition and algorithms for physical synthesis attached to it.

1) Characterizing a systolic array: Systolic arrays are typically resulting from nested loop parallelization, in various applicative domains (digital signal processing, DNA comparison, image processing, ...). Basically the corresponding architecture is a regular array of processors (either 1-dimensional or 2-dimensional), and each processor is dedicated to efficiently perform the body of the inner loop. All processors are only connected to their neighbors except the first and last processors that are connected to the "external world".

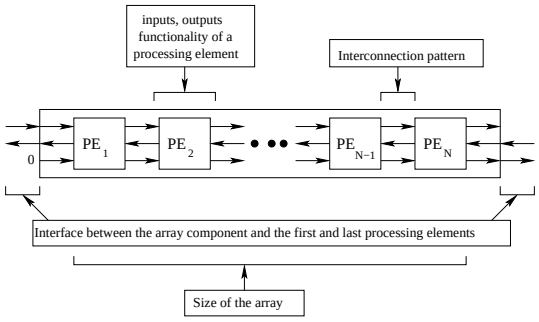

Fig. 7. Characterizing a systolic array

So, systolic arrays are one of the most structured and regular component that can be found. As seen on figure 8, a systolic array is described by a few number of characteristics :

- the inputs, outputs and functionality of one processing element, either structurally or behaviorally described
- the interconnection pattern between two neighbors processors
- the interface between inputs/outputs of the systolic array and inputs/outputs of the first and last processors (there could have inputs arbitrarily initialized or unused outputs)
- the size of the array

As it can be seen, having a DSL for describing systolic array permit to have a very concise description, allowing design flow tools to easily and rapidly access to essential data about the design.

2) Using structural properties in the design flow: As seen previously, regularity in a systolic array structure occurs mainly in the processors functionality and the interconnection pattern. As all processors have an identical structure, the results of a computation acting on one processor structure can be replicated for all processors. That's were specific algorithms can gain a lot comparing to a generic or flat design approach. Depending on limitations related to design tools and target technology, productivity gains could occur in each step of the design flow :

- synthesis, optimizations and mapping : whether the processor description is behavioral or structural, synthesis, optimization and mapping are just operating on one processor bounding box, reducing drastically the complexity of this step.
- placing : as for the previous step, complexity is reduced to placing one processor structure, then simultaneously replicating and floorplanning it for the whole array. Floorplanning complexity is lower than placing a flat design, since it act on coarser grain component, and that we have the capabilities to constraint the placement of one processor to a geometric shape (rectangular) easier to floorplan. Moreover the floorplanning is more or less automatically deduced from the systolic array topologies (that is to say 1D or 2D grid placement).
- routing : assuming that routing could be constrained to a bounding box, replicating the routing scheme of a processor can also be done. Finding one routing pattern between two neighbors processors and replicating it is not unrealistic if routing conflicts are overcome.

All these optimizations of design flow step induce a loss of computation complexity, thus decrease the design runtime and increases the productivity. Moreover, by mapping physically the systolic array structure to a reconfigurable unit, gain also includes increased clock frequency (by reducing wire length). As the presented methods do not allow sharing resources between neighbors processor, a loss of area can be feared against a flat design approach, but one should have in mind that we talk about large designs, and large reconfigurable units, so spending a lot of time and eventually not succeed in achieving physical synthesis to gain some area is not an issue.

3) A method to use structural properties for placement: Here we just show as example a concrete method to use systolic array structural properties for placing on a fine grain reconfigurable unit, trough the tool FRAP (FPGA Regular Array Placer) [8]. The aim of this tool is to put the maximum number of processing elements of a linear systolic array on a reconfigurable unit, given constraints about the locations of the first and last processor. To do so, it is used to add placement directives targeting a specific FPGA to a structural description of a systolic array. Finding such a placement acts in three steps :

1) All possible geometric shapes for a processing element are generated by combining all shapes of its sub-components
2) A full snake-like placement is determined using the processing element shapes previously computed
3) The final internal placement of the processing elements is performed according to their shapes

The second step is divided in two phases : (1) given location constraints for first and last processors, divide the reconfigurable grid into sub-area suitable to easier placement and (2) for each area, place a maximum number of processing elements in a snake-like fashion, using dynamic programming solving classical knapsack optimization problem.

The figure 8 shows a basic example of placing a systolic array with this method.

Gains resulting of using this tool for targeting particular FPGA circuits are various [9]. Placement step runtime is divided up to factor of 6. Routing step runtime is divided up to factor of 3. Clock frequency is increased up to a factor of

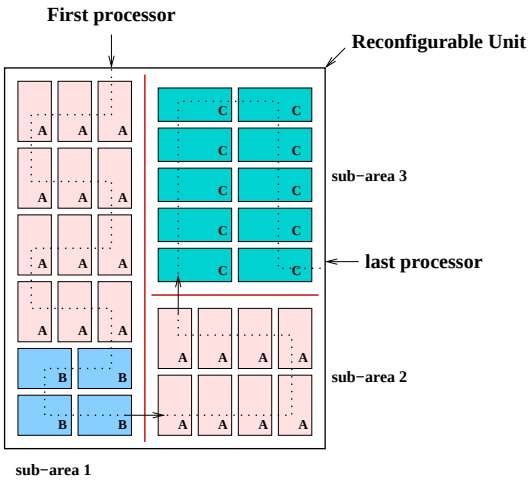

Fig. 8. placing a systolic array on reconfigurable unit using structural properties

2. However in some cases results are quite limited and even worst, principally due to lack of control over the vendor design tools that were used.

B. Why and how taking care of design structure and regularity ?

Given the previous example of using structural properties for physical synthesis, we can extent this principal to all structured components. Once described in a HLL, a circuit have a structure which is deduced from his DAG. The structure of a circuit can have various degrees of regularity, occurring at various hierarchical levels, ranging from low (identical slices of an adder) to high (identical processors of a regular array). It is even possible to extract structure and regularity from a flat design.

Recurrent questions about structure are : is it necessary to keep it throughout all the design flow ? and how to take advantages of keeping the structure information ?

When describing the advantages of keeping structure circuit in mind, one should make the difference between the advantages just induced by knowing the structure, and the advantage induced by having regularity in the structure.

1) the structure keep information about the interconnections (logical optimization, mapping, placement) without needing to recompute it at each design flow step. It principally permit to improve design density and frequency.
2) the regularity permit to reduce the execution time needed by the design flow, by factorizing tasks, that it to say just find a solution for a structural template and replicate it for all entities assimilated to this template. This method can be applied in the steps of synthesis, logical optimization, mapping, placement and routing, if the software or technological environment permits it.

From those 2 criterion, taking care of design structure increases density, frequency and decreases design flow running time.

As applicative design and reconfigurable unit area become larger, using structural properties will allow to deal with the increasing complexity of physical synthesis, although by the past this approach offers limited improvement and big effort to develop specific tools, due to the need to be adapted to closed vendor design environment.

VI. Conclusion

In the context of reconfigurable heterogeneous platforms, we are proposing a method allowing to produce components from productive development tools. These components can be synthesized for different compute units such as processors or fine grain FPGAs. We are actively working to rise the capabilities of synthesis tools to address mixed grain units. This implies the development of transformations such as loop unrolling, and support for architecture synthesis[14].

This approach has been made possible by the development of an open framework in which target reconfigurable architectures can be represented, with the immediate benefit of basic tools for physical design. The cellular automata example has actually been implemented in a past project with other components. The new tools created in the object-oriented environment are considerably easing developments, portability, modular assembly of components. The case of systolic arrays is significant in terms of physical design problems since these circuits can be described with simplicity, they produce a lot of computing power, and they are resource hungry.

While there is no reason to restrict the component design method to object-oriented languages and tools, it is expected that such environments will ease the management of run-time exchanges, application development and system activity description.

References

[1] L. Benini and G. DeMicheli. Networks on chip : a new paradigm for soc design. 2002.

[2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[3] K. Bouazza, J. Champeau, P. Ng, B. Pottier, and S. Rubini. Implementing cellular automata on the ArMen machine. In P. Quinton and Y. Robert, editors, *Proceedings of the Workshop on Algorithms and Parallel VLSI Architectures II*, pages 317–322, Bonas, France, June 1991. Elseiver.

[4] J. Cong and Y. Ding. Combinational logic synthesis for lut based fpga. *ACM transaction on DAES*, 1996.

[5] Florent de Dinechin. The price of routing in FPGAs. *Journal of Universal Computer Science*, 6(2) :227–239, February 2000.

[6] P. Dhaussy, J.-M. Filloque, B. Pottier, and S. Rubini. Global Control Synthesis for an MIMD/FPGA Machine. In *IEEE Workshop on FPGAs for custom computing machines (FCCM'94)*, pages 51–58, Napa, CA, April 1994.

[7] E. Caspi, M. Chu, R. Huang, J. Yeh, A. DeHon , and J. Wawrzynek. Stream computations organized for reconfigurable execution (score). In *FPL'2000*, 2000.

[8] E. Fabiani and D. Lavenier. Placement of linear arrays. In *FPL'2000 :10th International Workshop on Field Programmable Logic and Applications*, Villach, Austria, 2000.

[9] E. Fabiani and D. Lavenier. Experimental evaluation of place-and-route of regular arrays on xilinx chips. In *First International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, 2001.

[10] L. Lagadec. *Abstraction, modélisation et outils de CAO pour les circuits intégrés reconfigurables*. PhD thesis, Université de Rennes 1, December 2000.

[11] Loïc Lagadec, Bernard Pottier, and Oscar Villellas-Guillen. An lut-based high level synthesis framework for reconfigurable architectures. In *Synthesis, Architectures and Modeling of Systems (SAMOS 2)*, 2002.

[12] B. Lampson and K. Pier. The dorado, a high performance personal computer. 1981.

[13] G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti. In *Design systems for VLSI circuits*. Martinus Nijhoff Publishers, 1987.

[14] G. De Michelii. In *Synthesis and optimization of digital circuits*. Mc Graw Hill, 1994.

[15] J-D. Nicoud and A. Martin Tyrrell. The transputer t414 instruction set. *IEEE Micro*, 9(3), 1989.

[16] J. Oberg. Program : a grammar based method for specification and hardware synthesis of communication protocols, 1999.

[17] J. Oberg, A. Kumar, and A. Hemani. Scheduling of outputs grammar based synthesis of data communication protocols. *RIT, DEESD, Kista, Sweden.*, 1999.

[18] Alain Plantec. *Exploitation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code*. PhD thesis, Université de Rennes I, 1999.

[19] E.M. Sentovich and al. Sis : A system for sequential cirquit synthesis. Technical Report UCB/ERL M92/41, EECS, Berkeley, May 1992.

[20] D. Spinellis. Relaible softwre implementation using domain specific languages. 1999.

[21] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10), 1992.