A Flexible System Level Design Methodology Targeting Run-Time Reconfigurable FPGAs

Fabienne Nouvel, Florent Berthelot, Dominique Houzet

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 793919, 18 pages
doi:10.1155/2008/793919

Research Article

A Flexible System Level Design Methodology Targeting Run-Time Reconfigurable FPGAs

Florent Berthelot,[1] Fabienne Nouvel,[1] and Dominique Houzet[2]

[1] *CNRS UMR 6164, IETR/INSA Rennes, 20 avenue des Buttes de Coesmes, 35043 Rennes, France*
[2] *GIPSA-Lab, INPG, 46 avenue Felix Viallet, 38031 Grenoble Cedex, France*

Correspondence should be addressed to Florent Berthelot, florent.berthelot@irisa.fr

Received 31 May 2007; Revised 8 October 2007; Accepted 20 November 2007

Recommended by Donatella Sciuto

Reconfigurable computing is certainly one of the most important emerging research topics on digital processing architectures over the last few years. The introduction of run-time reconfiguration (RTR) on FPGAs requires appropriate design flows and methodologies to fully exploit this new functionality. For that purpose, we present an automatic design generation methodology for heterogeneous architectures based on DSPs and FPGAs that ease and speed RTR implementation. We focus on how to take into account specificities of partially reconfigurable components from a high-level specification during the design generation steps. This method automatically generates designs for both fixed and partially reconfigurable parts of an FPGA with automatic management of the reconfiguration process. Furthermore, this automatic design generation enables a reconfiguration prefetching technique to minimize reconfiguration latency and buffer-merging techniques to minimize memory requirements of the generated design. This concept has been applied to different wireless access schemes, based on a combination of OFDM and CDMA techniques. This implementation example illustrates the benefits of the proposed design methodology.

1. INTRODUCTION

Applications such as multimedia, encryption, or wireless communication require highly repetitive parallel computations and lead to incorporate hardware components into the designs to meet stringent performance and power requirements. On the other hand, the architecture flexibility is the key point for developing new multistandard and multiapplication systems.

Application-specific integrated circuits (ASICs) are a partial solution for these needs. They can reach high performance, computation density, and power efficiency but to the detriment of architecture flexibility as the computation structure is fixed. Furthermore, the nonrecurring engineering costs (NREs) of ASICs have been increased dramatically and made them not feasible or desirable for all the applications especially for bugfixes, updates, and functionality evolutions.

Flexibility is the processor's architecture paradigm. The algorithm pattern is computed temporally and sequentially in the time by few execution units from a program instruction stream. This programmability potentially occurs at each clock cycle and is applied to a general computation structure with a limited computation parallelism capacity. The datapath can be programmed to store data towards fixed memories or register elements, but cannot be truly reconfigured. This kind of architecture suffers from the memory controller bottleneck and their power inefficiency. These architectures have a very coarse-grained reconfiguration, reported as *system level*.

Reconfigurable architectures can provide an efficient platform that satisfy the performance, flexibility, and power requirements of many embedded systems. These kinds of architectures are characterized by some specific features. They are based on a spatial computation scheme with high parallelism capacity distributed over the chip. Control of the operator behavior is distributed instead of being centralized by a global program memory. Multiple reconfigurable architectures have been developed [1, 2]. They can be classified by their reconfiguration granularity.

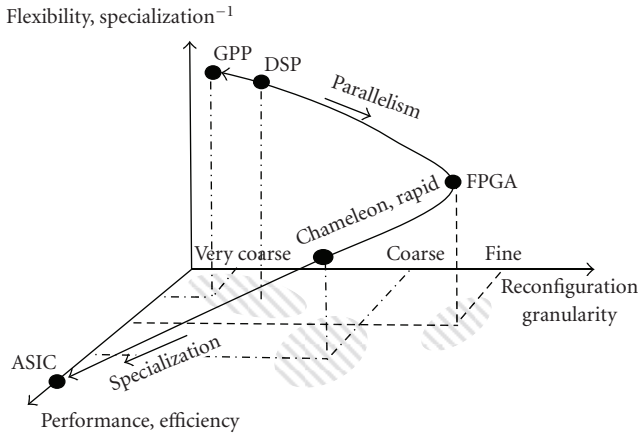


Figure 1: Architecture flexibility in regard with granularity and performance.

FPGAs have a logic level of reconfiguration. Communication networks and functional units are bit-level configurable.Their highly flexible structure allows to implement almost any application. A volatile configuration memory allows to configure the datapath and the array of configurable logic blocks.

Some reconfigurable architectures are based on coarse-grained arithmetic units of larger sizes such as 32 bits for algorithms needing word-width data paths, such as DART [3], Chameleon [4], Rapid [5], or KressArray [6]. They use a functional level of reconfiguration allowing to reach a greater power efficiency and computational density. Moreover, the amount of configuration data needed is limited. But coarse-grained architectures target domain-specific applications.

Figure 1 represents the architecture flexibility in regard with the granularity and performance of the above mentioned architectures. The architecture granularity elevation allows either specialization and performance or the improvement of architecture flexibility as for the processors.

The introduction of dynamically reconfigurable systems (DRSs) has opened up a new dimension of using chip area. Recently, run-time reconfiguration (RTR) of FPGA parts has led to the concept of virtual hardware [7].

RTR allows more sections of an application to be mapped into hardware through a hardware updating process. A larger part of an application can be accelerated in hardware in contrast to a software computation. Partial reconfiguration ability of recent FPGAs allows updating only a specified part of the chip while the other areas remain operational and unaffected by the reconfiguration. These systems have the potential to provide hardware with a flexibility similar to that of software, while leading to better performance. In [8] an extensive review of benefits of hardware reconfiguration in embedded systems is addressed. Some applications like software defined radios (SDR) [9] are from them. This emerging radio technology allows the development of multiband, multifunctional wireless devices [10].

However, cutting-edge applications require heterogeneous resources to efficiently deal with the large variety of signal and multimedia processing. This is achieved by mixing various processing granularities offered by general purpose processors (GPPs), digital signal processors (DSPs), and reconfigurable architectures. Methodologies based on a system-level design flow can handle these heterogeneous architectures. Our proposed design flow uses graphs to represent both the application and the architecture and aims at obtaining a near optimal scheduling and mapping of the application tasks on an heterogeneous architecture [11].

This paper presents our methodology allowing a fast integration and use of run-time reconfigurable components, especially FPGAs. This implies a convenient methodology and conception flow which allows a fast and easy integration of run-time reconfiguration onto applications and an automatic management of the reconfiguration process. The design space exploration will benefit from this improvement as operations of the application algorithm graph could be mapped either on DSP/GPP devices or statically/dynamically on FPGA devices thanks to run-time reconfiguration.

The rest of the paper is organized as follows. Following the introduction, Section 2 gives an overview of related approaches, especially those which deal with run-time reconfiguration from a high-level design specification. Section 3 introduces the system-level design methodology used in this work based on the CAD tool named SynDEx [12], which is used for the application partitioning stage of our methodology. Section 4 deals with the impact of run-time reconfiguration on this methodology and the way of modeling partially reconfigurable FPGAs from a system-level point of view. Section 5 then addresses the automatic design generation targeting run-time reconfigurable architectures. The overall design flow and the generated architecture layers are described. We detail the way to generate an automatic management of run-time reconfiguration, along with the steps to achieve design generation, memory minimization, and dynamic operator creation necessary for partial reconfiguration. These design flow and methodology have been applied in Section 6 for the implementation of a transmitter system for future wireless networks for 4G air interface. Two implementation examples, based on a network on chip (NoC) and a point to point communication scheme, are presented. Finally Section 7 provides a conclusion of this work and gives some indication of future works.

2. RELATED WORK

Numerous researches are focused on reconfigurable architectures and on the way to exploit efficiently their potentials. Higher level of abstraction, design space exploration, hardware/software partitioning, codesign, rapid prototyping, virtual component design and integration for heterogeneous architectures; all these topics are strong trends in architectures for digital signal processing.

This section reviews several relevant propositions in these fields with a special emphasis on frameworks which enable run-time reconfigurations of applications from a high-level design specification along with considerations of hardware-dependent steps allowing partial reconfiguration.

Silva and Ferreira [13] present a hardware framework for run-time reconfiguration. The proposed architecture is

based on a general-purpose CPU which is tightly connected to a dynamically reconfigurable fabric (DRF). A tool named BitLinker allows the relocation and assembly of bitstreams of individual components and is used to automate these steps which are very dependent on the underlying hardware's organization. This approach is architecture-centered and the application mapping steps on this specific platform are not addressed.

This issue requires a higher level of abstraction for the design specification. PaDReH [14] is a framework for the design and implementation of run-time reconfigurable systems and deals only with the DRS hardware design flow. The use of SystemC language enables a higher level abstraction for the design and validation. After a translation to the RTL level a space-time scheduling and hardware partitioning is performed allowing the generation of a run-time reconfiguration controller. Bitstreams generation is based on Xilinx [15] modular design flow. In [16], a technique based on some modifications of the SystemC kernel is presented. It allows to model and simulate partial and dynamic reconfiguration. The acquired results can assist in the choice of the best cost/benefit tradeoff regarding FPGA chip area.

Craven and Athanas [17] present a high-level synthesis (HLS) framework to create HDL. The hardware/software partitioning is performed by the designer. Reconfiguration simulations and reconfiguration controller generation is allowed from a modified version of the Impulse C ANSI C-based design language supporting HLS from C to RTL HDL. Nevertheless, the final implementation steps are not addressed.

The EPICURE project [18] is a more comprehensive methodology framework based on an abstraction tool that estimates the implementation characteristics from a C-level description of a task and a partitioning refinement tool that realizes the dynamic allocation and the scheduling of tasks according to the available resources in the dynamically reconfigurable processing unit (DRPU). An intelligent interface (ICURE) between the software unit and the DRPU acts as a hardware abstraction layer and manages the reconfiguration process. Overlapping between computations and reconfigurations is not supported.

In [19], Rammig and Al present an interesting tool-assisted design space exploration approach for systems containing dynamically hardware reconfigurable resources. A SystemC model describes the application, while an architecture graph models the architecture template. Next a multi-objective evolutionary algorithm is used to perform automatic system-level design space exploration. A hierarchical architecture graph is used to model the mutual exclusion of different configurations.

Most of these above methodologie frameworks assume a model of external configuration control, mandating the use of a host processor or are tightly coupled to a specific architecture [13, 18]. Hence custom heterogeneous architectures are not supported. Few of them address methods for specifying run-time reconfiguration from a high-level down to the consideration of specific features of partially reconfigurable components for the implementation.

Our proposed methodology deals with heterogeneous architectures composed of processors, FPGA or any specific circuits. The implementation of run-time reconfiguration on hardware components, especially FPGAs, is automated and eased by the use of a high-level application and architecture specification. This step is handled by the SynDEx tool which allows the definition of both the application and the hardware from a high level and realizes an automated Hardware/software mapping and scheduling. Run-time reconfiguration and overall design control scheme are independent of any specific architecture.

3. GENERAL METHODOLOGY FRAMEWORK

Some partitioning methodologies based on various approaches are reported in the literature [11, 20, 21].

They are characterized by the granularity level of the partitioning, targeted hardware, run-time-reconfiguration support, on-line/off-line scheduling policies, HW-SW relocation [22], reconfiguration prefetching technique [23] and flow automation.

As discussed in the previous section, few tools based on these partitioning methodologies provide a seamless flow from the specification down to the implementation.

To address these issues, this work uses the SynDEx CAD tool for the high-level steps of our methodology, which include automatic design partitioning/scheduling and RTL code generation for FPGA.

3.1. AAA/SynDEx presentation

Our methodology aims at finding the best matching between an algorithm and an architecture while satisfying constraints. AAA is an acronym for adequation algorithm architecture, adequation is a French word meaning efficient matching. AAA methodology is supported by SynDEx, an academic system-level CAD tool and is used in many research projects [24, 25]. AAA methodology aims at considering simultaneously architecture and application algorithm, both are described by graphs, to result in an optimized implementation.

The matching step consists in performing a mapping and a scheduling of the algorithm operations and data transfers onto the architecture processing components and the communication media. It is carried out by a heuristic which takes into account durations of computations and inter-component communications to optimize the global application latency. Operation durations and inter-component communications are taken into account for that purpose.

3.1.1. Application algorithm graph

Application algorithm is represented by a dataflow graph (DFG) to exhibit the potential parallelism between operations. Figure 2 presents a simple example. The algorithm model is a direct data dependence graph. An operation is executed as soon as its inputs are available, and this DFG is infinitely repeated. SynDEx includes a hierarchical algorithm representation, conditional statements and iterations of algorithm parts. The application can be described in

a hierarchical way by the algorithm graph. The lowest hierarchical level is always composed of indivisible operations (C2, F1, and F2). Operations are composed of several input and output ports. Special inputs are used to create conditional statements. Hence an alternative subgraph is selected for execution according to the conditional entry value. As illustrated in Figure 2, the conditional entry "X" of "C1" operation represents such a possibility. Data dependencies between operations are represented by valued arcs. Each input and output port has to be defined with its length and data type. These lengths are used to express either the total required data amount needed by the operation before starting its computation or the total amount of data generated by the operation on each output port.

3.1.2. Architecture graph

The architecture is also modeled by a graph, which is a directed graph where the vertices are computation operators (e.g., processors, DSP, FPGA) or media (e.g., OCB busses, ethernet) and the edges are connections between them. So the architecture structure exhibits the actual parallelism between operators. Computation vertices have no internal computation parallelism available. An example is shown in Figure 3. In order to perform the graph matching process, computation vertices have to be characterized with algorithm operation execution times. Execution times are determined during the profiling process of the operation. The media are also characterized with the time needed to transmit a given data type.

3.1.3. AAA results

The result of the graph matching process is a mapping and scheduling of the application algorithm over the architecture. These informations are detailed by the automatically generated files called "macrocodes." Macrocodes are generated for each computation vertex of the architecture graph. Figure 4 shows a possible macrocode file which describes the behavior of the vertex named "FPGA1" when only the "C2" operation is mapped on. The macrocode is a high-level language made of hardware independent primitives. Macrocodes are always composed of several threads. There is one thread for the computation controls where operations are called, like the C2 operation with input and output buffers.

There is one thread for each communication port of the operator as modeled in the architecture graph. Communication threads can receive and send data thanks to the "Recv_" and "Send_" primitives. The primitives "Suc1_," "Suc0_," "Pre0_," "Pre1_" manipulate the semaphores to ensure mutual exclusions and synchronizations between threads.

There are also macros for memory allocations. They define depth and data type of buffers used to store the computation results of the operator (number of generated events on the arc defined in the application algorithm graph). The set of generated macrocode files represents a distributed synchronized executive. This executive is dedicated to the application and can be compared to an offline static operating system.

Figure 5 depicts the overall methodology flow. Each macrocode is translated toward a high-level language (HDL or C/C++) for each HW and SW component. This translation produces an automatic dead-lock free code. The macrocode directives are replaced by their corresponding code from libraries (C/C++ for software components, VHDL for hardware components). Many libraries have been developed for heterogeneous platforms and we present how we extend SynDEx capacities to handle runtime reconfigurable components.

4. RUN-TIME RECONFIGURATION CONSIDERATIONS

4.1. Architecture graph modeling of run-time reconfigurable components

In our methodology, the algorithm graph application description is realized at the functional level representing coarse-grained operations. This description is handled during the design flow through the use of libraries containing IP definitions for code generation. These coarse-grained IPs are supposed to be developed to fully exploit parallelism of their final implementation targets. This step is basically achieved by the compiler for software components or the synthesizer for hardware components. Hence, the logical-level architecture granularity of FPGA is exploited by the RTL-synthesizer to generate IP netlists.

With this high-level modeling, static components (which do not provide reconfiguration support) and run-time reconfigurable components are modeled by vertices in the architecture graph. We have defined two kinds of vertices to match reconfiguration support of hardware components: the *static vertices* and the *run-time reconfigurable vertices*. Run-time reconfigurable vertices can be considered as an extension of static vertices in term of functional flexibility as their operations can be changed dynamically.

As the last FPGA generations have reached high densities, it can be advantageous to divide them in several independent parts for architecture modeling. Example (b) in the left side of Figure 6 presents the modeling of an FPGA divided in five parts. One is considered as static (F1), two represent embedded microprocessors (P1, P2), and the last two represent two partially reconfigurable areas (D1, D2). We can also see the architecture graph modeling of this FPGA made with several vertices under SynDEx. The internal links named CC1, IL1, IL2 and IL3 represent data communications between these sub-FPGA parts. The modeling of a prototyping platform (case (a), Figure 6) composed of two FPGAs, one CPLD and a processor, can be represented with the same architecture graph. Only the characterization of the communication media differs.

The top of Figure 6 shows a set of six operations composing an algorithm graph. Once the mapping process is done, these operations are affected to the architecture graph vertices. All operations mapped to a same vertex are sequentially executed at run-time. Hence we have defined three kinds of operators for the operation executions on the hardware. The first is a *static operator* which can not be reconfigurated, so dedicated to one operation. The two others, named

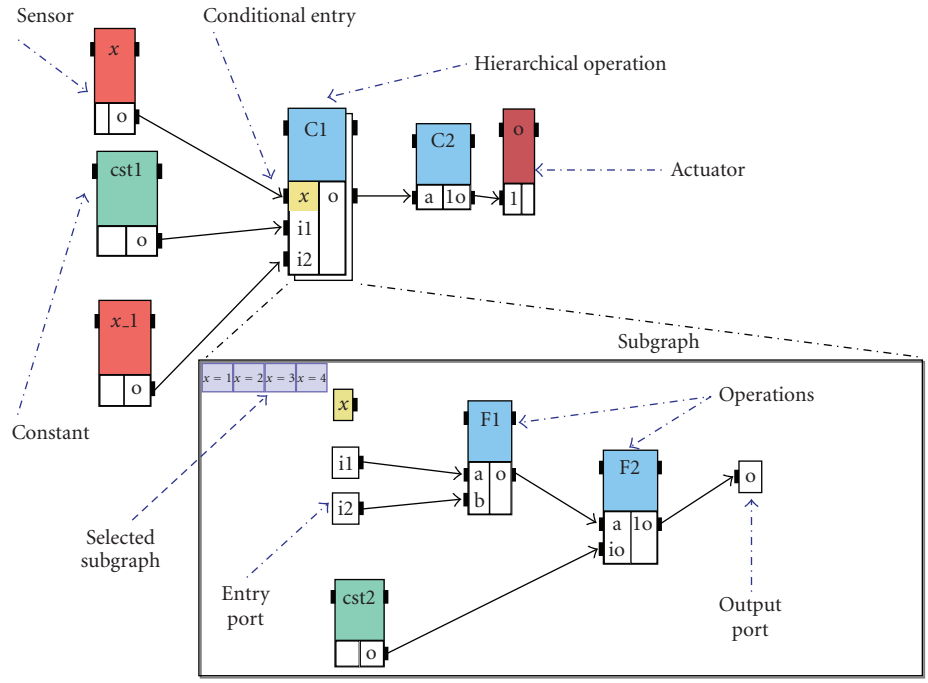


Figure 2: Algorithm graph.

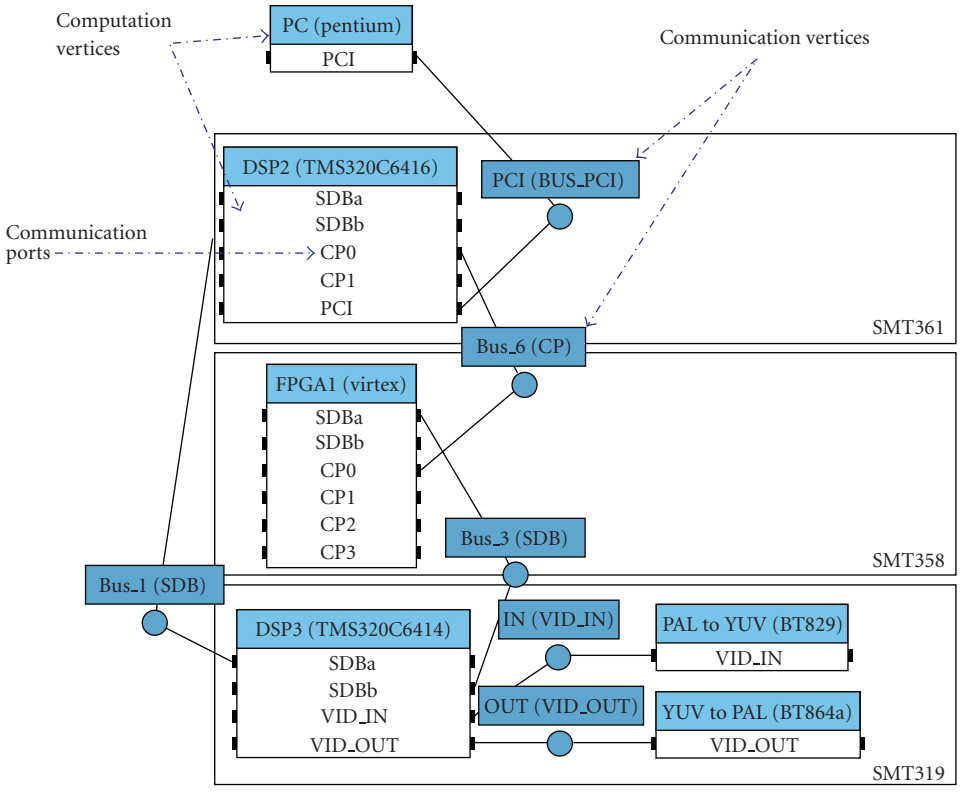


Figure 3: Architecture graph of a prototyping platform.

```
Include (syndex.m4x)
Processor_(Virtex, FPGA1)
 Semaphores_(                          }
 mem1_empty, mem1_full,                } Semaphore
 mem2_empty, mem2_full,                } allocation
            )                          }
Alloc_(char, mem1, 92)                 }   Memory allocation
Alloc_(char, mem2, 92)                 }  (data type, name, depth)
Thread_(CP, Bus6)
      Loop_
          Suc1_(mem1_empty)            }
          Recv_(mem1 , bus6)           }
          Pre0_(mem1_full)             }  Communication
          Suc1_(mem2_full)             }       thread
          Send_(mem2 , bus6)           }
          Pre0_(mem2_empty)            }
Endloop_
Endthread_
Main_
Loop_
          Suc0_(mem1_full)             }
          C2(mem1, mem2)               }  Computation
          Pre1_(mem1_empty)            }      thread
          Pre1_(mem2_full)             }
Endloop_
Endmain_

Endprocessor_
```

Figure 4: Simple SynDEx macrocode example generated for a computation vertex.

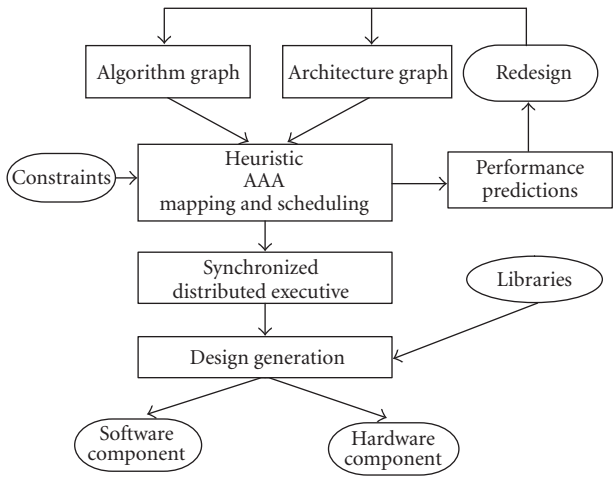


Figure 5: SynDEx methodology flow.

parameterizable operator and *dynamic operator*, are able to dynamically change their functionalities to match a given algorithm graph operation. They differ by their ways to implement a new operation during the runtime.

We define "*dynamic operator*" as a special operator which is managed to implement sequentially new operations through run-time reconfiguration. On current FPGAs it can be achieved thanks to partial reconfiguration technology and is mainly dependent on the specific architecture features. The *Parameterizable operator* is appropriate to enable fast operation switching through a parameter selection step.

Unlike to static vertex, a reconfigurable vertex can implement some static operators, some parameterizable operators and some dynamic operators. As a result, additional functionalities to control run-time reconfigurations in order to implement dynamically new functionalities, are necessary.

By assigning parts of the design from a high level, we can make run-time reconfiguration specification more flexible and independent of the application coding style. Moreover that allows to keep this methodology flow independent of the final implementation of the run-time reconfiguration process which is device dependent and tightly coupled to backend tools. This modeling can be applied on various components of different granularities.

4.2. Operation selection criterions for a dynamic implementation

Selection of operations for a dynamic implementation is a multicriteria problem. Reconfiguration can be used as a mean to increase functionality density of the architecture or for flexibility considerations. However some general rules can be brought out. The boundary between a fixed or a dynamic implementation is depending on the architecture ability to provide a fast reconfiguration process. This point is generally linked to the reconfiguration granularity of the architecture. When a set of common coarse grain operators are used for the execution of several operations, the use of a parameterized structure could be a more efficient solution and could allow fast inter-operation switchings [26]. That is achieved by increasing the operator granularity. As a result the amount of data configurations needed is decreased.

Parametrization and partial reconfiguration concepts are closely linked. In both cases the aim is to change the functionality, only the amount of configuration data and their targets differ. From the manager point of view both operators share a same behavior.

The choice of implementation style (static, partial reconfiguration, parameterized structure) of operations can be based on the ratio between reconfiguration time of operator, computation time and activity rate. A good mapping and scheduling is a tradeoff to avoid that reconfigurable surfaces remain unused or under exploited.

To illustrate this, Figure 7 represents a set of eight operations composing an application. These operations can be ordered in a diagram. Operations which are placed closely have a high degree of commonality and nearly the same complexity (logical resources). Hence a parameterized operator can be created, as for the case of operations B, C, and D. This parameterized operator does not have any configuration latency. Operations A, E, F, and H have nearly the same degree of complexity but they doesn't share a great amount of commonality and have not a high activity rate. In this case they can be implemented by a dynamic operator with run-time reconfiguration. Operation G has a high activity rate, it is desirable to implement it as a static operator.

The number of operations implemented on a same dynamic operator has a direct impact on the implementation

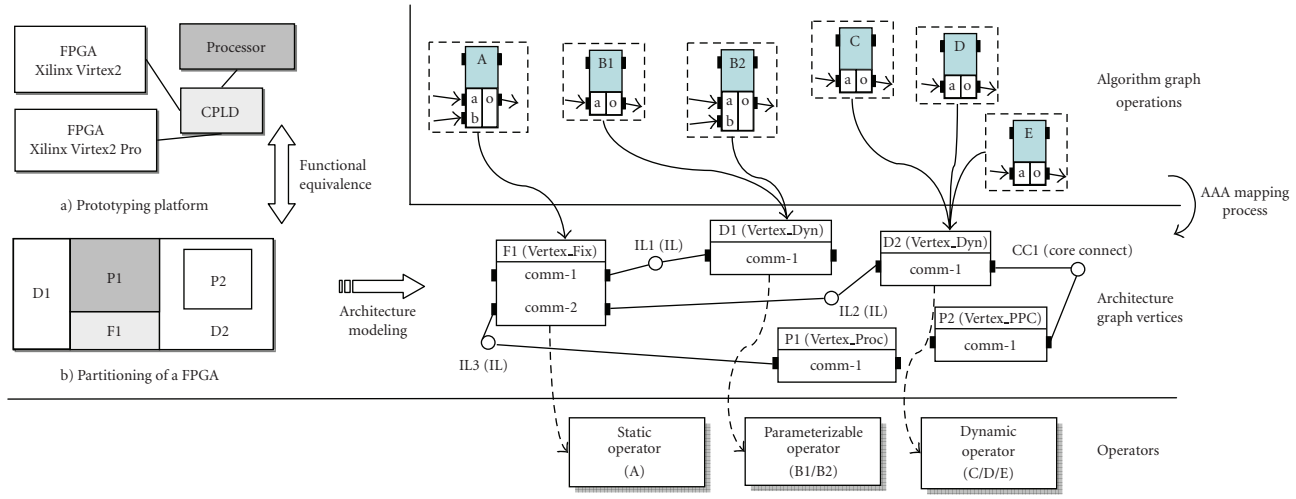


Figure 6: Architecture graph example and vertex operations implementation style.

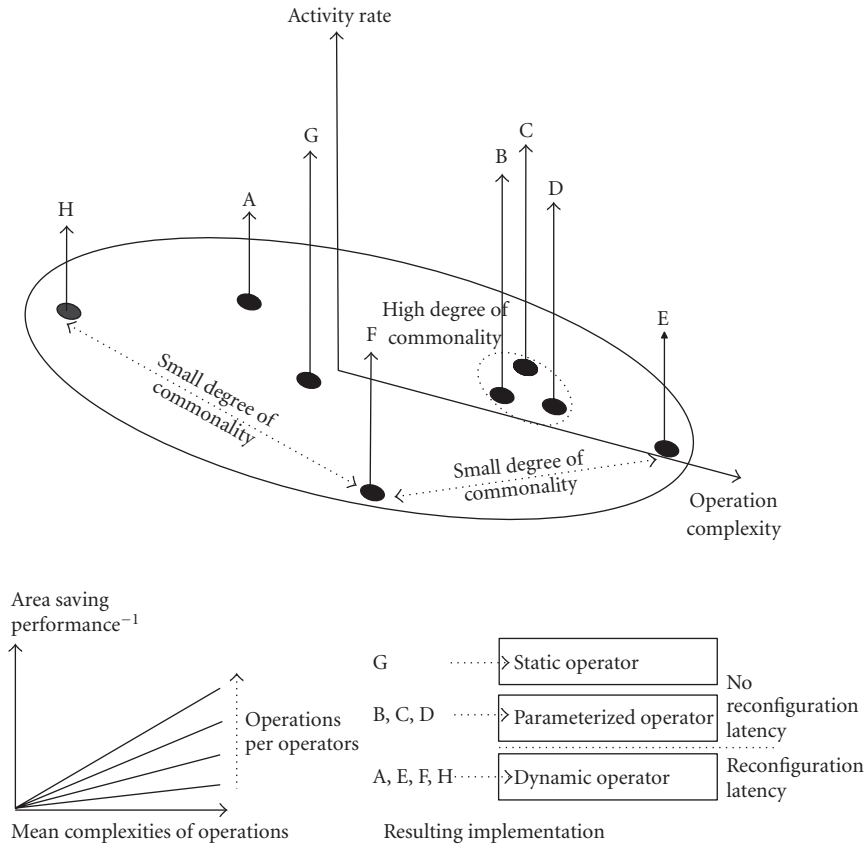


Figure 7: Choice and influence of the implementation style of operations.

area and the performance. The extreme cases are those when all the operations are implemented through only one dynamic operator or on the contrary when each operations are static and consume more resources. The influence of the number of operations per operator over the saving space is decreasing when the mean complexity of the operations are decreasing. So, we can consider that dynamic reconfiguration is useful when applied to medium and complex operations which do not have high activity rates, thus avoiding to waste too much time in reconfiguration. On the other hand, operations of small complexities can remain static or parameterized.

As the heuristic does not yet support run-time reconfigurable vertices (reconfiguration latencies are not taken into

account), the operations of the algorithm graph have to be characterized to define their way of implementation (static or dynamic). It is achieved by a manual mapping constraint from the designer through SynDEx tool.

We plan to automate this step thanks to the extraction of some metrics from the algorithm graph such as the operation complexity, activity rate, and relative commonality. These metrics could be used to guide the designer in the static-dynamic functionality implementation tradeoff.

4.3. Minimizing the reconfiguration cost

Run-time partially reconfigurable components must be handled with a special processing during the graph matching process. A major drawback of using run-time reconfiguration is the significant delay of hardware configuration. The total runtime of an application includes the execution delay of each task on the hardware combined with the total time spent for hardware reconfiguration between computations. The length of the reconfiguration is proportional to the reprogrammed area on the chip. Partial reconfiguration allows to change only a specified part of the design hence decreasing the reconfiguration time. An efficient way to minimize reconfiguration overhead is to overlap it as much as possible with the execution of other operations on other components. It is known as configuration prefetching techniques [27].

Therefore the heuristic of SynDEx has to be improved to take into account reconfiguration overhead with configuration prefetching techniques to minimize it. A solution based on a genetic heuristic is under development to integrate more metrics during the graph matching process. This study applied on run-time reconfiguration is presented in [28]. However the architectural model considered is based on a processor and a single-context partially reconfigurable device which acts as a coprocessor. The ideal graph matching process is a completely automated selection of operation implementation styles (fixed, dynamic, parameterized) while satisfying multiple constraints.

5. AUTOMATIC DESIGN GENERATION

5.1. General FPGA synthesis scheme

The overall design generation flow and hierarchical design architecture are shown in Figure 8. For each vertex of the architecture graph,after the mapping and scheduling processes, a synchronized executive represented by a macrocode is generated.

A list defines the implementation way of each algorithm graph function. The following cases are possible.

(i) For static vertices, only static operators are allowed.
(ii) For reconfigurable vertices, static operators, parameterizable operators, and dynamic operators can be implemented.

Macrocodes generated for these vertices must be handled by specific libraries during the design generation to include dynamic reconfiguration management. The design generation is based on the translation of macrocodes to the VHDL code. This translation uses the GNU macro processor M4 [29] and macros defined in libraries in addition to communication media and IP operators. Each macrocode is composed of one computation thread to control the sequencing of computations, and communication threads for each independent communication port. These threads are synchronized through semaphores. This high-level description has a direct equivalence in the generated design and corresponds to the highest design level which globally controls datapaths and communications. A lower level is in charge of resources management, typically in case of simultaneous accesses on buffers for computations or communications. The *configuration manager* process belongs to this design level. The next level brings together resource controls. Resources are operators, media and buffers. The *protocol builder* functionality, which is detailed later, has access to the underlying configuration memory.

A same operator can be reused many times over different data streams but only one instantiation of each kind of operator appears in VHDL. We have defined a standard interface for each operator through encapsulation, as described in the next section.

These points lead to build complex data paths automatically. Our libraries can perform these constructions by using conditional VHDL signal assignments. In next sub-sections, we deal with some important issues: optimization of memory for exchanged data through buffer merging, operators merging for dynamic instantiation and design generation for run-time reconfiguration management.

5.2. Macrocode preprocessing for memory minimization

SynDEx is based on a synchronous dataflow model of computation, processes communicating through buffers and reading/writing a fixed number of tokens each time they are activated. Hence this kind of representation allows to determine completely the scheduling at compile-time. Applications often have many possible schedules. This flexibility can be used to find solutions with reduced memory requirements or to optimize the overall computation time. Memory minimization is essential for an implementation on reconfigurable hardware devices which have limited on-chip memory capabilities.

Currently only the latency optimization is taken into account during the matching step of SynDEx. Macrocodes are generated without memory minimization considerations. The latest SynDEx version operates a macrocode memory minimization analysis on the macrocodes. A graph coloring based heuristic is used to find which buffers have to be merged. Buffers allocated by SynDEx can be heterogeneous as they store computation results of operations working on various data depths and widths. The integrated memory minimization of SynDEx allows to merge buffers of various depth and same data widths. In the other cases a manual analysis must be made by the designer to merge buffers of different widths. Table 1 sums up the cases.

The result is a list of buffers which must be merged into a global one. Hence, global buffers are automatically generated

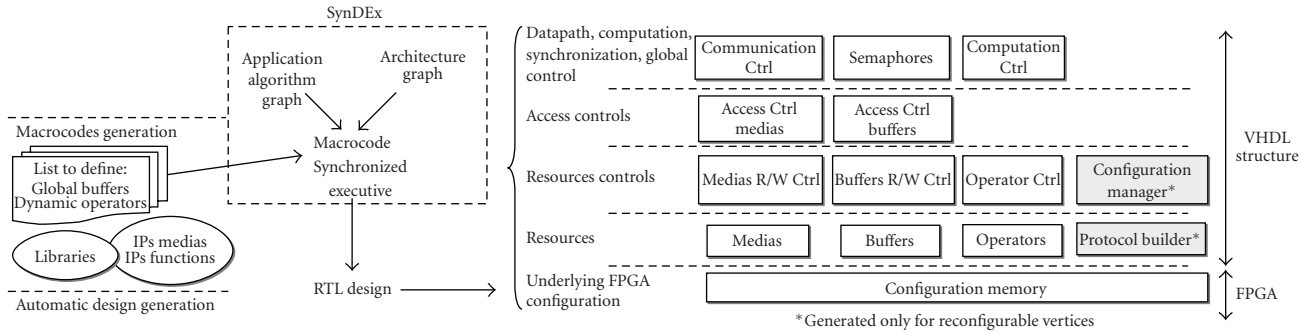


Figure 8: Hierarchical architecture design generation for hardware components.

Table 1: Buffer minimization cases.

Depth	Width	
	Same	Different*
Same	SynDEx	Manual
Different*	SynDEx	Manual

*Buffers widths must be multiple.

according to the operator data types. We have the following denotations:

(i) $L : \{b_1, b_2, \ldots, b_n\}$: *A list of n buffers which can be merged, where* $L_{(k)} = b_k$,

(ii) D_k: *the depth of buffer k.*

(iii) W_k: *the data width of buffer k.*

Hence the total amount of memory needed is

(i) without buffer merging: $M_{\overline{\text{bm}}} = \sum_{i=1}^{n} D_{L(i)^*} W_{L(i)}$,

(ii) with buffer merging: $M_{\text{bm}} = \max(D_{L(i)^*} W_{L(i)})$ for $1 \leq i \leq n$.

So, the amount of memory saved is

$$S_{\text{mem}} = M_{\overline{\text{bm}}} - M_{\text{bm}}. \quad (1)$$

Global buffers are automatically generated in VHDL. We have developed a parameterizable memory IP able to work on various depths and data widths and providing fast accesses. This IP is implemented either on dedicated hardware (i.e., Xilinx blockram) or on distributed on-chip FPGA memory (look-up tables).

5.3. Generic computation structure

After selection of candidates among all the operators of the algorithm graph for partial reconfiguration or parameterization, we obtain a set of operators that must be implemented into a run-time reconfigurable device. To achieve design generation, a generic computation structure is employed. This structure is based on buffer merging technique and functionality abstraction of operators. The aim is to obtain a single computation structure able to perform through run-time reconfiguration or parameterization the same functionalities as a static solution composed of several operators.

Encapsulation of operators through a standard interface allows us to obtain a generic interface access. This encapsulation eases the IP integration process with this design methodology and provides functionality abstraction of operators. This last point is helpful to easily manage reconfigurable operators with run-time reconfiguration or configuration for parameterized operators.

This encapsulation is suitable for coarse-grained operators with dataflow computation. It is a conventional interface composed of an input data stream, an output data stream along with "enable" and "ready" signals to control the computation. In the case of parameterized operators, a special input is used to select the configuration of the operator.

As operators can work on various data widths, the resulting operator interface has to be scaled for the most demanding case. Next, we have to apply our buffer merging technique, which allows us to create global buffers able to work on various depths and data widths, as presented in the previous section.

Figure 9 presents such a transformation operated on two operators (Op_A and Op_B), working on different data widths and depths. A generic reconfigurable operator named *Op_Dyn* is created and its interface is adapted. Its functionality will be defined through run-time reconfiguration by the *configuration manager* process. Only the *Config_Select* input is added in the case of a parameterizable operator (*Op_Param*).

5.4. Design generation for run-time reconfiguration management

In order to perform reconfiguration of the dynamic part we have chosen to divide this process in two subparts: a *configuration manager* and a *protocol builder*. The *"configuration manager"* is automatically generated from our libraries according to the sequencing of operations expressed in the macrocode. A *"configuration manager"* is attached to each parameterizable or dynamic operator.

The configuration manager is in charge of operation selection which must be executed by the configurable operator by sending configuration requests. These requests are sent only when an operation has completed its computation and

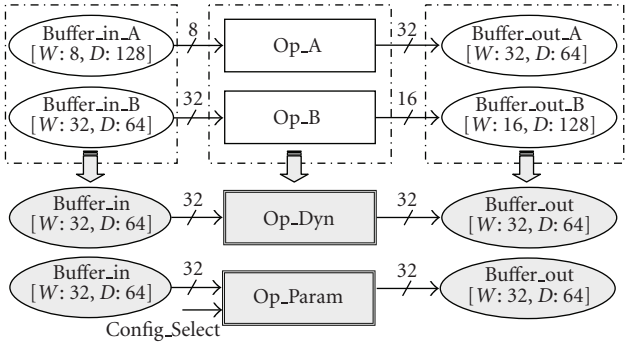


Figure 9: Buffers and operators merging example.

if a different operation has to be executed after. So reconfigurations are performed as soon as the current operation is completed in order to enable configuration prefetching as described before. This functionality provides also information on the current state of the operator. This is useful to start operator computations (with *"enable"* signal) only when the configuration process is ended.

Figure 10 shows a simple example based on two operations (j and k) which are executed successively. Labels **M** and **P** show where the functionalities *"configuration manager"* and *"protocol builder"* are, respectively, implemented.

Case (a) shows the design generated for a nonreconfigurable component. The two operators are physically implemented and are static. Case (b) is based on a parameterizable operator; the selection of configurations is managed by the *configuration manager*. There is no configuration latency; the operator is immediately available for computation. The signal *Config_Select* is basically a request of configuration, it results in the selection of a set of parameters among all internally stored. The third case (case (c)) is based on a dynamically reconfigurable operator which implements successively the two operations thanks to run-time reconfiguration. The reconfigurable part provides a virtual hardware, so at a time only one operator is physically implemented on this dynamic part. The configuration requests are sent to the *protocol builder* which is in charge to construct a valid reconfiguration stream in agreement with the used protocol mode (e.g., selectmap).

The configuration manager can perform reconfiguration (parametrization or partial reconfiguration) during other communications or computations scheduled on the same vertex. If the next configuration is conditional, this one is performed as soon as its value is known, allowing reconfiguration prefetching.

Encapsulation of operators with a standard interface allows to reconfigure only the area containing the operator without altering the design around. Buffers and functionalities involved in the overall control of the dynamic area remain on a static part of the circuit. This partitioning allows to reduce the size of the bitstream which must be loaded and decreases the time needed to reconfigure.

5.5. Reconfiguration control implementation cases

This way of proceeding must be adapted according to architecture considerations. There are many ways to reconfigure partially an FPGA. Figure 11 shows five solutions of architectures for this purpose.

Case (a) shows a standalone self-reconfiguration where the fixed part of the FPGA reconfigures the dynamic part. This case can be adapted for small amounts of bitstream data which can be stored in the on-chip FPGA memory. However, bitstreams require often a large amount of memory and cannot fit within the limited embedded memory provided by FPGAs, so bitstreams are stored in an external memory as depicted in case (b).

Case (c) shows the use of a microprocessor to achieve the reconfiguration. In this case, the FPGA sends reconfiguration requests to the processor through hardware interrupts, for instance. This microprocessor can be considered as a slave for the FPGA. The CPLD is used to provide some "glue logic" to link these two components. In case (c), the FPGA is still the initiator of the reconfiguration requests but they are performed through a microprocessor. This case is probably the most time consuming as the "protocol builder" functionality is a task on the microprocessor which can be activated through an hardware interrupt. Hence a noticeable latency is added, which is due to the program loading and fetching and hardware/software handshake.

In case (d), the FPGA can be seen as a slave as the reconfiguration control is fully managed by the microprocessor. In such a case the FPGA is used as a coprocessor for hardware acceleration. Last FPGA generations have embedded microprocessors and allow the last case of reconfiguration control. Implementation of these functionalities have a direct impact on the reconfiguration latency.

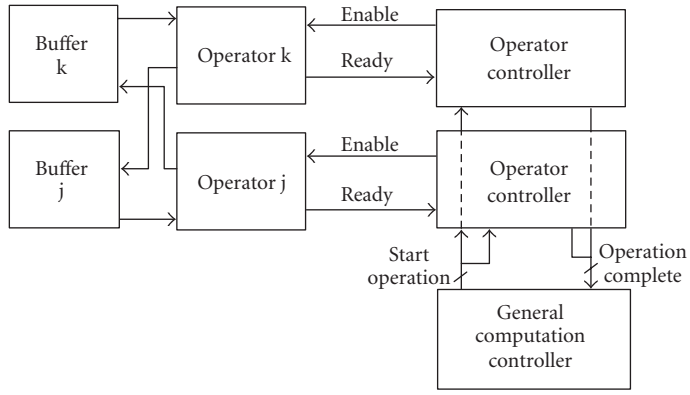
6. IMPLEMENTATION EXAMPLE

Our design flow and methodology have been applied for the implementation of a transmitter for future wireless networks in a 4G-based air interface [30]. In an advanced wireless application, SDR does not just transmit. It receives informations from the channel networks, probes the propagation channel and configures the system to achieve best performance and respond to various constraints such as bit-error rate (BER) or power consumption. We have considered here a configurable transmitter which can switch between three transmission schemes.

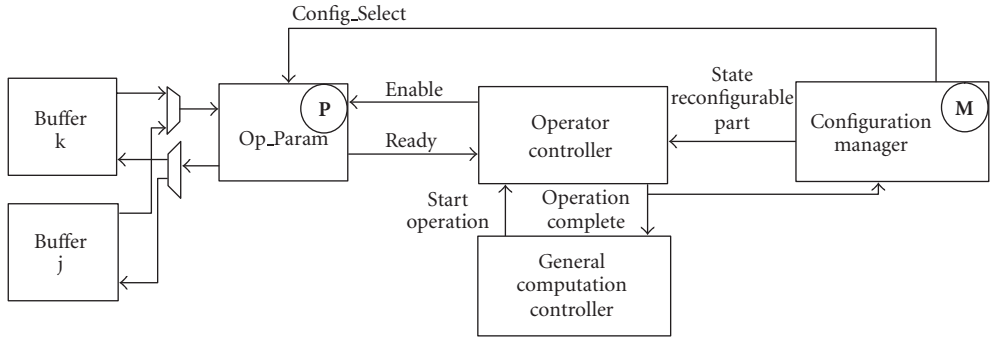
The basic transmission scheme is a multicarrier modulation based on orthogonal frequency division multiplexing (OFDM). OFDM is used in many communications systems such as: ADSL, Wireless LAN 802.11, DAB/DVB, or PLC. The first scheme corresponds to the most simple transmitter configuration named OFDM.

The second configuration uses a multicarrier code division multiple access (MC-CDMA) technique [30]. This multiple access scheme combines OFDM with spreading allowing the support of multiple users at the same time.

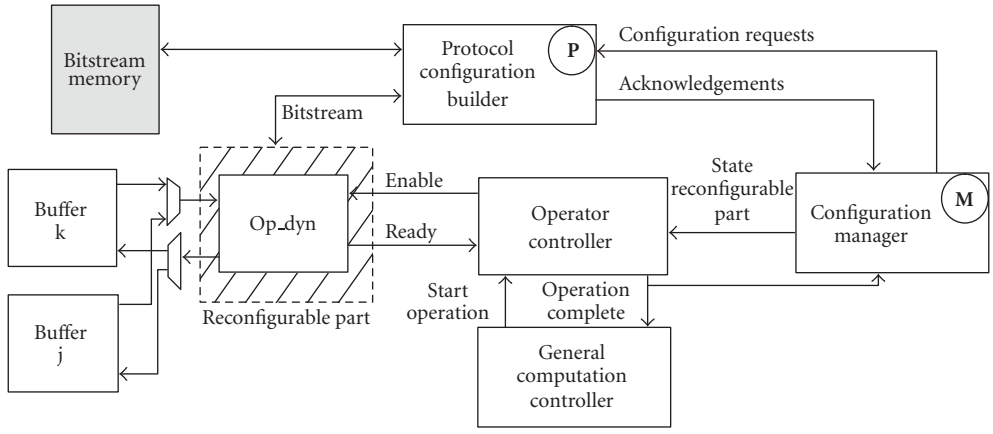
The third configuration uses a spread-spectrum multicarrier multiple access with frequency hopping pattern



(a) Static operator based structure



(b) Parameterizable operator-based structure



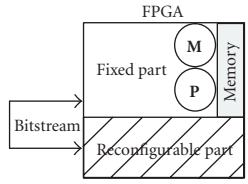
(c) Dynamic operator-based structure

Figure 10: Architecture comparison between a fixed/parameterized or dynamic computation-based structure.

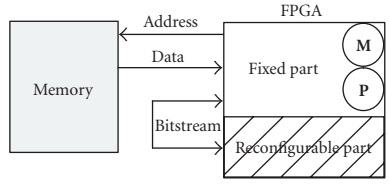
(FH-SS-MC-MA), as used in 802.11b standard (WiFi). It is a spread spectrum modulation technique where the signal is repeatedly switching frequencies during radio communication, minimizing probability of jamming/detection.

SynDEx algorithm graph, depicted by Figure 12(a), shows the numeric computation blocks of this configurable multimode transmitter.

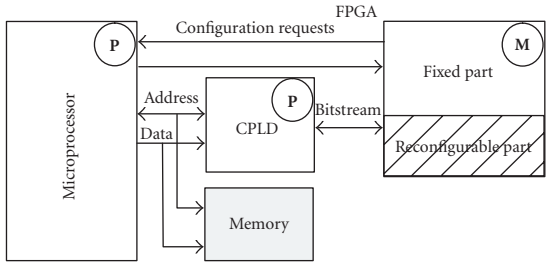
These three transmission schemes use channel coding and perform a forward correction error (FEC), corresponding to the *Channel_coding* block. The DSP can change the FEC to select a Reed-Solomon encoder or a convolutional encoder. Next, a modulation block performs an adaptive modulation between QPSK, QAM-16, or QAM-64 modulations. For MC-CDMA and FH-SS-MC-MA schemes, a *spreading* block implements a fast Hadamard transform (FHT). This block is inactive for OFDM scheme. A chip mapping (*Chip_mapping* block) is used in order to take into account the frequency diversity offered by OFDM modulation. This block performs either an interleaving on OFMD symbols for MC-CDMA, whereas the interleaving in FH-SS-MC-MA



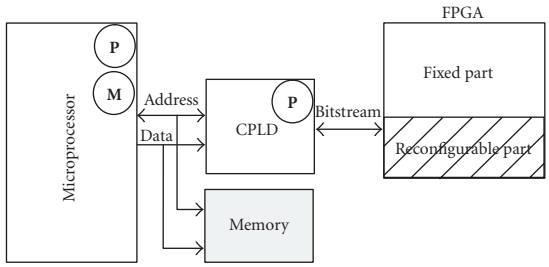
(a) Standalone self reconfiguration with on-chip bitstream memory



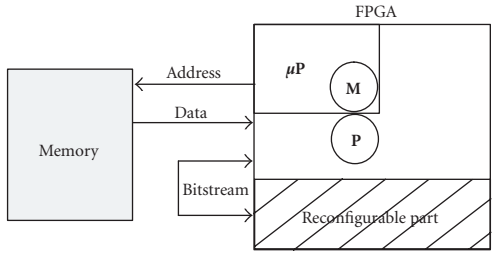
(b) Standalone self reconfiguration with off-chip bitstream memory



(c) Reconfiguration through a microprocessor



(d) Microprocessor reconfiguration control (FPGA as a slave)



(e) Standalone self-reconfiguration with embedded microprocessor

Figure 11: Different ways to reconfigure dynamic parts of an FPGA.

scheme is a frequency hopping (FH) pattern to allow the user data to take advantage of the diversity in time and frequency. The OFDM modulation is performed by an inverse fast Fourier transform thanks to *IFFT* block which also implements zero-padding process. For complexity and power consumption this IFFT can be implemented in radix-2 (*IFFT-R2*) or radix-4 (*IFFT-R4*) mode. Configuration selection (conditional entry *Config*) and data generation are handle by the *Data_Gen* block, whereas *DAC_If* block represents the interface to the digital-to-analog converter (DAC) device of the platform.

6.1. Implementation on a prototyping platform

We have implemented this reconfigurable transmitter on a prototyping board from Sundance Technology (Matignon, France) [31]. This board is composed of one DSP (C6701 from Texas Instrument) and one partially reconfigurable FPGA (Xc2v2000 from Xilinx with 10752 slices). Its SynDEx representation is shown in Figure 12(b). Communications between DSP and FPGA are ensured by SHB (Sundance high-speed bus) and CP (communication Port) communication medium from Sundance Technology. We have chosen to divide the FPGA in four vertices. One is static (*Interface*) and represents pre-developed logic for FPGA interfacing. The three remaining vertices (*FPGA_Dyn*, *FPGA_Dyn_1*, and *FPGA_Dyn_2*) are run-time reconfigurable vertices. Internal communications between these parts are ensured by the LI media.

Table 2 details the configurations and complexities of the reconfigurable transmitter computational blocks (depending on the transmission schemes). These complexities are obtained on a Xilinx VirtexII FPGA, where each slice includes two 4-input function generators (LUT). Some of these functions can be implemented thanks to the available Xilinx IPs [32].

6.1.1. Functions mapping

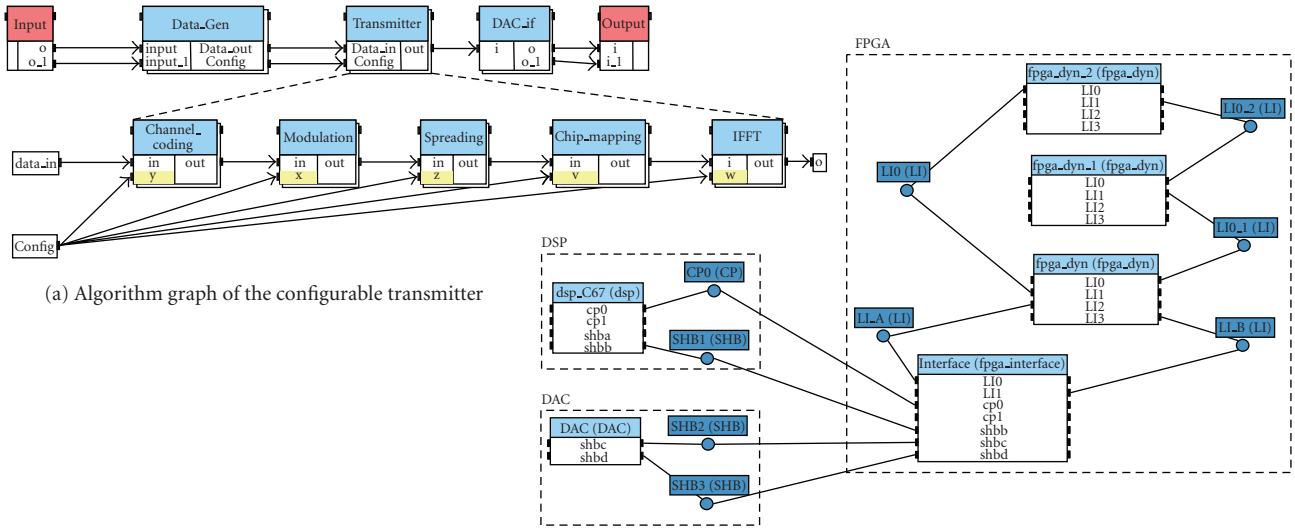
From the characterization of the computational blocks we can determine a possible implementation of the transmitter on the prototyping board. Table 3 summarizes this mapping.

IFFT block will be implemented thanks to a parameterizable IP from Xilinx, and mapped on the *FPGA_Dyn* vertex. Two static operators used for the Reed-Solomon encoder and convolutional encoder are also mapped on *FPGA_Dyn* vertex.

Functionalities *Interleaving* and *FH* of the *Chip mapping* block will be sequentially implemented by a dynamic operator with run-time reconfiguration on the *FPGA_Dyn_1* vertex.

Spreading block (*FHT*) will be performed through a static operator implemented on the *FPGA_dyn_2* vertex. On the same vertex the modulation is a parameterizable operator.

The *Data_Gen* and *DAC_If* blocks are mapped on the *Interface* vertex to ensure DSP's data and configurations transmission. They will not be detailed as we focus on the generation for the run-time reconfigurable vertices.



(a) Algorithm graph of the configurable transmitter

(b) Prototyping board architecture graph

Figure 12: Algorithm and architecture graphs of the configurable transmitter.

Table 2: Configurations and complexities.

Transmitter configuration	Computational blocks	Channel coding		Modulation		Spreading		Chip mapping		IFFT	
(1) Sampling frequency = 20 Mhz (2) Number of users = 32 (3) FFT = 256 points (4) OFDM symbol duration = 12.8 us (5) Frame duration = 1.32 ms	Configurations and complexities	A B C	Reed-Solomon encoder: 120 slices.	A B C	QPSK, 16QAM, 64QAM	B C	FHT (spreading factor: 32): 873 slices.	A B	Interleaving: 186 slices, 2 BlockRam.	A B C	IFFT-R2 (256-points, 16 bits): 752 slices, 3 BlockRam, 3 Mult18*18 – IP Xilinx COREGen xFFT v3.1.
		A B C	Convolutional encoder: 43 slices Xilinx Convolution Encoder v5.0.	A B C	Parameterizable operator: 60 slices			C	Frequency Hopping Pattern (FH): 246 slices, 2 Block-Ram.	A B C	IFFT-R4 (256-points, 16 bits): 1600 slices, 7 BlockRAM, 9 Mult18*18 – IP Xilinx COREGen xFFT v3.1. Parameterizable operator: 1600 slices - IP Xilinx COREGen xFFT v3.1.

Transmission schemes: OFDM (A), MC-CDMA (B), FH-SS-MC-MA (C).

6.1.2. *Resulting FPGA architecture*

The code, both for the fixed and dynamic parts, has been automatically generated with SynDEx thanks to the libraries. However, the generation of bitstreams needs a specific flow from Xilinx called modular design [15]. Modular design is based on a design partitioning in functional modules which are separately implemented and allocated to a specific FPGA area. Each module is synthesized to produce a netlist and then placed and routed separately. Reconfigurable modules communicate with the other ones, both fixed and reconfigurable, through a special bus macro (3-state buffers or slice-based bus macros) which is static. They guarantee that each time the partial the reconfiguration is performed the routing channels between modules remain unchanged. Partial bitstreams are created from the individual module designs.

Case (b) of Figure 11 represents our architecture. Virtex II integrates the ICAP FPGA primitive. ICAP is an acronym for internal configuration access port providing a direct access to the FPGA configuration memory and so enables a self-partial reconfiguration.

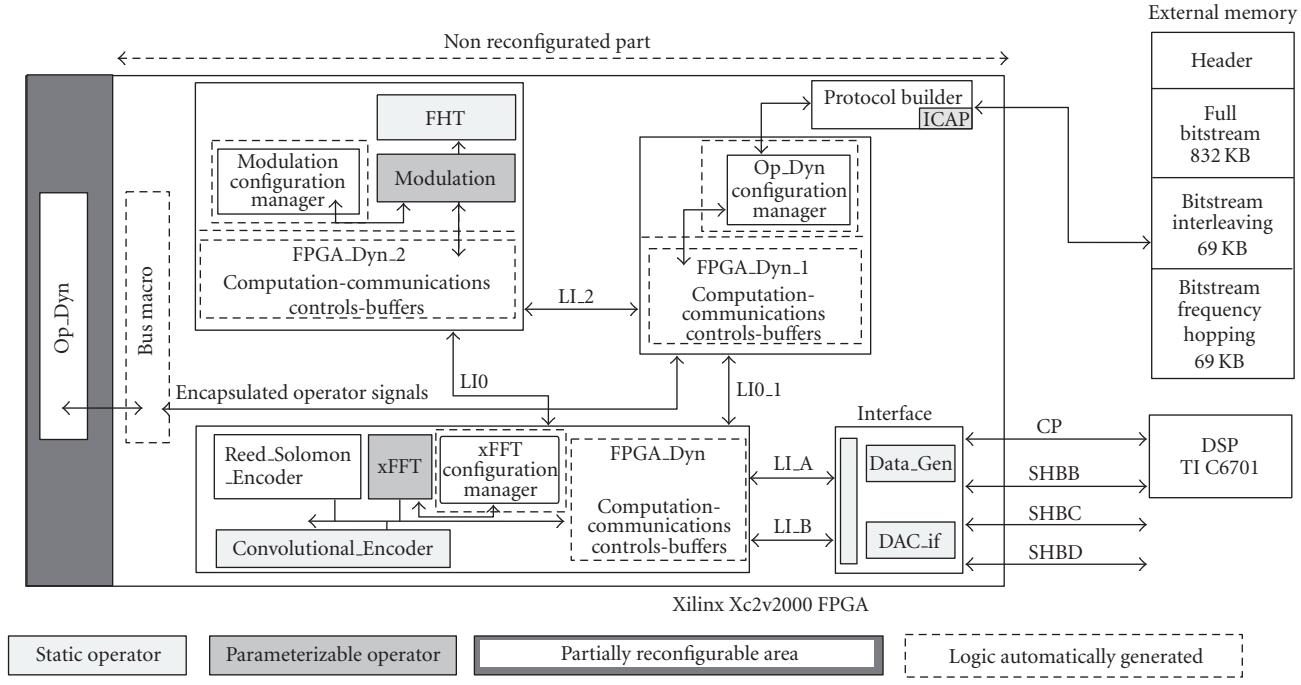
Figure 13 shows the resulting design of the reconfigurable transmitter which is compliant with the modular design flow for partial reconfiguration.

The nonreconfigurable part of the FPGA is composed of four areas resulting from the design generation of the four architecture graph vertices. Reconfigurable vertices architectures are detailed. Each one is composed of a general computation/communication controller with buffers.

Table 3: Functional blocks mapping (design automatically generated).

Computation vertice	FPGA_Interface	FPGA_Dyn	FPGA_Dyn_1	FPGA_Dyn_2	DSP C67	DAC
Functional Block	Data_Gen (1) DAC_if (1)	IFFT (2) Reed Solomon Encoder(1) Convolutional Encoder (1)	Chip Mapping (3)	FHT (1) Modulation (2)	Input	Output

(1): Static operator, (2): parameterizable operator, (3): dynamic operator.



CP : low speed digital communication bus for transmitter configuration
SHB : high speed digital communication bus for data transmission
ICAP : internal configuration access port

Figure 13: Reconfigurable transmitter architecture.

Parameterizable and run-time reconfigurable operators have their own configuration manager. The Dynamic operator configuration manager can address reconfiguration requests to the protocol builder. The protocol builder performs partial reconfigurations thanks to the ICAP primitive and bitstreams stored in an external memory (*interleaving* and *frequency hopping* bitstreams).

Only the left FPGA side is a run-time reconfigurable area and implements the dynamic operator. Encapsulated signals of the dynamic operator are accessed through bus macros as circumscribed by the modular Design flow. The size of the reconfigurable area has to be scaled to the most demanding function in logical resources, here FH function (246 slices, 2 BlockRam). Besides the shape of the reconfigurable area is constrained by the modular design and leads to allocate a greater area size than really necessary. In this case, the area takes the full FPGA height and 12 slices width (1300 slices). This area is the only run-time reconfigurated, other areas remain unchanged and are defined once during the full FPGA configuration.

Recently, this area constraint has been removed by the early-access partial reconfiguration design environment (EAPR) [32], which now allows partial reconfiguration modules of any rectangular size. This new modular design flow for run-time reconfiguration is supported by the PlanAhead floorplanner tool [33]. The VHDL files automatically generated by SynDEx are input source files for such floorplanning tools. Hence placement of bus macros, modules floorplanning, and partial bitstreams generation are performed with this target specific design environment. Nevertheless, our high-level methodology is independent of these backend steps concerning the final physical implementation. Any module-based flow for partial reconfiguration is compatible with our methodology.

6.1.3. Numerical results of implementation

The reconfiguration operates at 50 Mhz. The first and full configuration of the device takes 16 milliseconds while the partial reconfiguration process of chip mapping functionality (operator *Op_Dyn*) is about 2 milliseconds. That is of the order of several data frames, thus partial reconfiguration is suitable for a transmission scheme switching, as in the case of chip mapping functionality which is changed for MC-CDMA and FH-SS-MC-MA schemes. On the other hand, partial reconfiguration is too time consuming to be used for intratransmission scheme reconfiguration. It is the case if the channel coding and IFFT are implemented on the same dynamic operator.

6.1.4. Implementation comparison of chip mapping

As shown in Table 4, chip mapping operation is implemented either using "Interleaving" or "Frequency hopping" algorithms. Both have different complexities when implemented separately and statically (resp, 186 and 246 slices). With a dynamic implementation a same FPGA area is allocated for both (12% of the device) and each version requires 69 KB of external memory for the partial bitstreams (EA-PR flow [32] could reduce these needs).

FPGA resources needed to implement logic controls of the chip mapping functionality are more important with a dynamic reconfiguration implementation scheme (107 + 550 = 657 slices) as for a static and hand-made implementation (200 slices). The overhead is about 450 slices to allow run-time reconfiguration of the chip mapping functionality. This overhead is due to the generic VHDL structure generation, based on the macrocode description. However, this gap is decreasing with a greater number of configurations implemented on the same dynamic operator. The aim is to take advantage of the virtual hardware ability of the architecture. However, the flexibility and implementation speed up, through the automatic VHDL generation given by this methodology, can overcome this hardware resource overhead. For instance, we can add a Turbo convolutional encoder for the channel coding block (1133 slices, 6 BlockRam—IP Xilinx3GPP Compliant Turbo Convolutional Codec v1.0). As the size of the reconfigurable part is fixed by the designer, any design able to be satisfied with this area constraint can be implemented.

6.2. Implementation based on a network on chip

We have implemented the previous application on a specific architecture topology based on a Network on Chip. The goal is to combine the functional flexibility given by the run-time reconfiguration with the regular and adaptable communication scheme of an NoC. Hence allowing us to skip the need to redesign the architecture graph when the application is modified.

Network on Chip (NoC) is a new concept developed since several years and intended to improve the flexibility of IP communications and the reuse of intellectual property (IP) blocks. NoCs provide a systematic, flexible and scalable framework to manage communications between a large set of IP blocks. It can also reduce IP connection wires and optimize their usage. The dynamic reconfigurability of the communication paths responds to the fluctuating processing needs of embedded systems.

Dataflow IPs can be connected either through point to point links or through a NoC. Tools have been proposed in order to design and customize NoCs according to their application needs. We have developed both a NoC and its corresponding tool. This NoC is one possible target of the presented methodology. This NoC is adapted and optimized to allow the plug and the management of dynamically reconfigurable IPs. Reconfigurability is one important source of flexibility when combined with a flexible communication mechanism.

6.2.1. MicroSpider NoC presentation

Our NoC [34] is built with routing nodes using a wormhole packet switching technique to carry messages. Operators are linked to the routing nodes through network interfaces (NIs) with an FIFO-like protocol. Our NoC is customizable through an associated CAD tool [34]. Our CAD Tool is a decision and synthesis tool to help designers to obtain the hadhoc NoC depending on the application and implementation constraints. It is able to configure the various functionalities of our NoC. Finally, this tool generates an optimized dedicated NoC VHDL code at RTL level.

Network interfaces

Network interfaces are flexible and configurable to be adapted with the connected processing elements. They implement the network protocol. NIs connect IP blocks to the NoC. For NoC standardization reasons, we made the choice of the OCP interface [35] for the communication between NI and IPs. NI uses a table to transform OCP addresses map in packet header routing information according to the NoC configuration. The network interface architecture is strongly related to SynDEx scheduling technique that requires a virtual channel controller and buffering capabilities.

6.2.2. Implementation

The operations from the previous application example (Figure 12(a)) are dataflow operations. Dataflow operators have FIFO like protocols. We have added specific features to our NoC in order to optimize the dataflow traffic and the plug of IPs. We have also standardized the interfaces of the NoC. A subset of OCP interface standard has been selected and implemented.

We have implemented the previous application on a six-node NoC (Figure 14) integrated in the same FPGA, with one reconfigurable area per NoC routing node, and one node dedicated to a bridge to the external C6701 DSP. Each of the five remaining nodes can be the target of any application task. Several tasks can be grouped on the same node either to be dynamically reconfigured, parameterized or fixed and simply scheduled in time. We have evaluated latency and

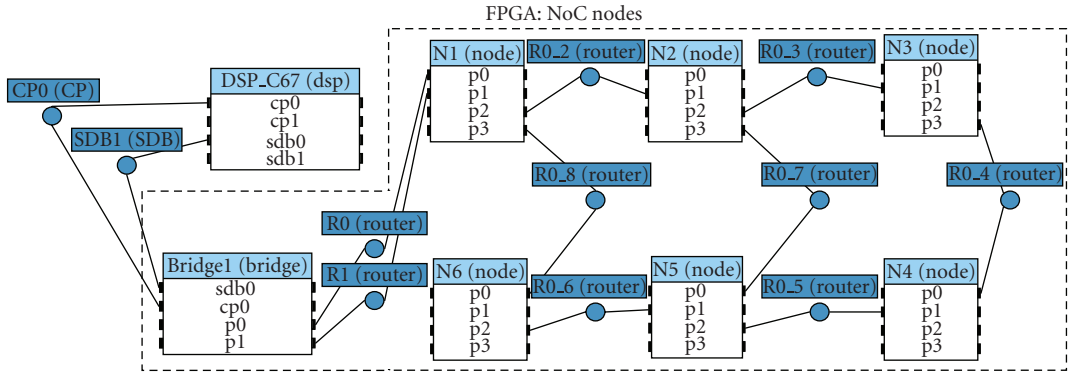


Figure 14: Architecture graph of the NoC nodes and the DSP.

Table 4: Static-dynamic implementation comparison of chip mapping.

Chip mapping functionalities coding and implementation	Designer (hand made) implementation: all static		Designed and generated automatically with SynDEx implementation: run-time reconfiguration		
	Controls	IPs (Interleaving+FH)	Controls		IPs
			Protocol Builder	Overall dynamic part controls	Reconfigurable area capacity (used)
Slices:	200	186 + 246 = 432	107	550	1300 (246)
Block RAM(18 Kbits):	2	—	—	2	14 (2)
FPGA area:	1.8%	4%	1%	5.1%	12% (2.5%)
Switching latency:	—	—	—	—	2 ms
External memory:	832 KB (Full bitstream)		832 KB (Full bitstream) + 2*69 KB (Partial bitstreams)		

Table 5: (1): Static operator, (2): parameterizable operator, (3): dynamic operator application function mapping on the NoC.

N1	N2	N3	N4	N5	N6
Turbo encoder (1)	Reed Solomon encoder(3) Convolutional encoder(3)	Modulation(2)	Spreading(1)	Chip mapping (3)	IFFT (2)

throughput of unloaded NoC links. These figures are introduced in SynDEx heuristic. Table 5 shows the functions mapping on this NoC As SynDEx schedules transfers in time, we use virtual channels in order to guarantee priority of first scheduled transfers. Thus the latency is deterministic and accurate. The M4 code generated by our methodology provides all the scheduling of treatments and communications as well as the source and target of each communication. These informations are extracted and translated to a C code [36] for a Xilinx Picoblaze micro-controller in charge of the dynamic operators and parameterizable operators.

Figure 15 details a NoC node structure. There is one Picoblaze for each NoC interface linked to dynamically reconfigurable operators. The Picoblaze controls the scheduling of operators, the size, the target and the starting of data transfers from the running operator. A NoC IP is the grouping of the operators, the picoblaze processor and the control operators and OCP adaptation logic. The scheduling of communications is managed with virtual channels in the NoC interfaces. They are configured by the Picoblaze. Implementation results are presented in Table 6.

Table 6: Noc node resources usage.

IP	Nb Slices	Freq (MHz)	Nb BRAM
PicoBlaze	110	200	1
NoC Node	430	200	2

The NoC cost is similar to the point to point solution with all the advantages of flexibility and scalability. With this solution there is no need to design a dedicated architecture graph for each new application. One general purpose 2D mesh can be selected for the architecture graph. Also, the coupling of an NoC with dynamically reconfigurable operators allow a new level of flexibility and optimization.

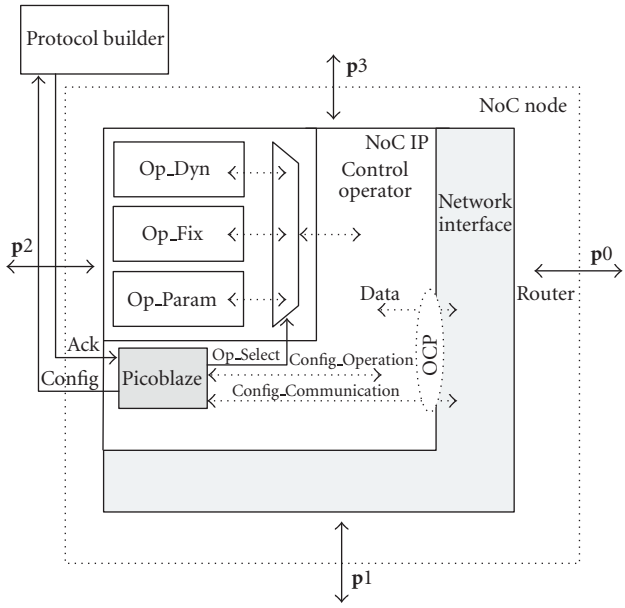


Figure 15: NoC Node detailed view.

7. CONCLUSION

We have described a methodology flow to manage automatically partially reconfigurable parts of an FPGA. It allows to map applications over heterogeneous architectures and fully exploit the advantages given by partially reconfigurable components. This design flow has the main advantage to target software components as well as hardware components to implement complex applications from a high-level functional description.

This methodology is independent of the final implementation of the run-time reconfiguration which is device dependent and achieved with back-end tools. This modeling can be applied on various components of different granularities. The AAA methodology and associated tool SynDEx have been used to perform the mapping and code generation for fixed and dynamic parts of FPGA. However, SynDEx's heuristic needs additional developments to fully take into account the reconfiguration time during the graph matching process.

That will allow the user to find a mapping and a scheduling of the application in order to improve the reconfiguration time or functional density of the resulting architecture. This methodology can easily be used to introduce dynamic reconfiguration on predeveloped fixed designs as well as for fast IP block integration on fixed or reconfigurable architectures. Thanks to the automatic code generation the development cycle is alleviated and accelerated. This top-down design approach makes it possible to accurately evaluate system implementation, according to functions complexity and architecture properties. Besides, the benefits of this approach fit into the SoftWare radio requirements for efficient design methods, and adds more flexibility and adaptation capacities through partial run-time reconfiguration on FPGA-based systems.

REFERENCES

[1] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[2] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 642–649, Munich, Germany, March 2001.

[3] R. David, D. Chillet, S. Pillement, and O. Sentieys, "Mapping future generation mobile telecommunication applications on a dynamically reconfigurable architecture," in *Proceedings of IEEE International Conference on Acoustic, Speech, and Signal Processing (ICASSP '02)*, vol. 4, p. 4194, Orlando, Fla, USA, May 2002.

[4] B. Salefski and L. Caglar, "Re-configurable computing in wireless," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 178–183, Las Vegas, Nev, USA, June 2001.

[5] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1436–1448, 2004.

[6] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable kressarrays," in *Proceeding of the 9th International Workshop on Field Programmable Logic and Applications (FPL '99)*, pp. 385–390, Glasgow, Scotland, August-September 1999.

[7] S. Erdogan, M. Eads, and T. Shaneyfelt, "Virtual hardware management for high performance signal processing," in *Proceedings of the 3rd IASTED International Conference on Circuits, Signals, and Systems (CSS '05)*, pp. 36–39, Marina del Rey, Calif, USA, October 2005.

[8] P. Garcia, M. Schulte, K. Compton, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP Journal of Embedded Systems*, vol. 2006, Article ID 56320, 19 pages, 2006.

[9] J. Mitola III, "Software radio architecture evolution: foundations, technology tradeoffs, and architecture implications," *IEICE Transactions on Communications*, vol. E83-B, no. 6, pp. 1165–1173, 2000.

[10] "Joint tactical radio system website," http://enterprise.spawar.navy.mil.

[11] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 335–340, Anaheim, Calif, USA, June 2005.

[12] C. Sorel and Y. Lavarenne, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of the 1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pp. 123–132, Mont Saint-Michel, France, June 2003.

[13] M. L. Silva and J. C. Ferreira, "Support for partial run-time reconfiguration of platform FPGAs," *Journal of Systems Architecture*, vol. 52, no. 12, pp. 709–726, 2006.

[14] E. Carvalho, N. Calazans, E. Brião, and F. Moraes, "PaDReH—a framework for the design and implementation of dynamically and partially reconfigurable systems," in *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design (SBCCI '04)*, pp. 10–15, Pernambuco, Brazil, September 2004.

[15] "Xapp290: two flows for partial reconfiguration: Module based or difference based," http://direct.xilinx.com/bvdocs/appnotes/xapp290.pdf.

[16] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. Melcher, "Modelling and simulation of dynamic and partially reconfigurable systems using systemc," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 35–40, Porto Alegre, Brazil, March 2007.

[17] S. Craven and P. Athanas, "A high-level development framework for run-time reconfigurable applications," in *Proceedings of the 9th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD '06)*, Washington, DC, USA, September 2006.

[18] J. P. Diguet, G. Gogniat, J. L. Philippe, et al., "EPICURE: a partitioning and co-design framework for reconfigurable computing," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 367–387, 2006.

[19] F. Dittmann, F.-J. Rammig, M. Streubühr, C. Haubelt, A. Schallenberg, and W. Nebel, "Exploration, partitioning and simulation of reconfigurable systems," *Information Technology*, vol. 49, no. 3, p. 149, 2007.

[20] B. Steinbach, T. Beierlein, and D. Fröhlich, "Uml-based codesign for run-time reconfigurable architectures," in *Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specifications from FDL '03*, pp. 5–19, Norwell, Mass, USA, 2004.

[21] Y. Qu, J.-P. Soininen, and J. Nurmi, "Static scheduling techniques for dependent tasks on dynamically reconfigurable devices," *Journal of Systems Architecture*, vol. 53, no. 11, pp. 861–876, 2007.

[22] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 986–991, Munich, Germany, March 2003.

[23] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '02)*, pp. 187–195, Monterey, Calif, USA, February 2002.

[24] V. Fresse, O. Déforges, and J.-F. Nezan, "AVSynDEx: a rapid prototyping process dedicated to the implementation of digital image processing applications on multi-DSP and FPGA architectures," *EURASIP Journal on Applied Signal Processing*, vol. 2002, no. 9, pp. 990–1002, 2002.

[25] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, O. Deforges, and Y. Sorel, "Rapid prototyping for heterogeneous multicomponent systems: an MPEG-4 stream over a UMTS communication link," *EURASIP Journal on Applied Signal Processing*, vol. 2006, Article ID 64369, 13 pages, 2006.

[26] A. Al Ghouwayel, Y. Louët, and J. Palicot, "A reconfigurable architecture for the FFT operator in a software radio context," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '06)*, pp. 181–184, Island of Kos, Greece, May 2006.

[27] J. Resano, D. Mozos, D. Verkest, S. Vernalde, and F. Catthoor, "Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL '03)*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 585–594, Lisbon, Portugal, September 2003.

[28] J. Harkin, T. McGinnity, and L. Maguire, "Modeling and optimizing run-time reconfiguration using evolutionary computation," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 661–685, 2004.

[29] "Gnu m4—macro processor," http://www.gnu.org/software/m4/.

[30] S. Le Nours, F. Nouvel, and J.-F. Hélard, "Design and implementation of MC-CDMA systems for future wireless networks," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 10, pp. 1604–1615, 2004.

[31] "Sundance multiprocessor technology ltd," http://www.sundance.com.

[32] "Xilinx intellectual property center," http://www.xilinx.com/ipcenter/.

[33] "Early access partial reconfiguration user guide, ug208 (v1.1)," Xilinx Inc, Tech. Rep., March, 2006.

[34] "Planahead," http://www.xilinx.com.

[35] S. Evain, J.-P. Diguet, and D. Houzet, "A generic CAD tool for efficient NoC design," in *Proceedings of International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS '04)*, pp. 728–733, Seoul, Korea, November 2004.

[36] "Programming FPGA's as MicroControllers: MicroFpga," http://www.microfpga.com/joomla.