An Auto-Adaptative Reconfigurable Architecture for the Control

Nicolas Ventroux, Stéphane Chevobbe, Frédéric Blanc, Thierry Collette

An Auto-Adaptative Reconfigurable Architecture for the Control

Nicolas VENTROUX, Stéphane CHEVOBBE, Fréderic BLANC and Thierry COLLETTE

CEA-List DRT/DTSI/SARC
Image and Embedded Computers Laboratory
F-91191 Gif-Sur-Yvette - FRANCE
phone: (33) 1-69-08-66-37
contact: firstname.surname@cea.fr

Abstract. Previous works have shown that reconfigurable architectures are particularly well-adapted for implementing regular processing applications. Nevertheless, they are inefficient for designing complex control systems. In order to solve this drawback, microprocessors are jointly used with reconfigurable devices. However, only regular, modular and reconfigurable architectures can easily take into account constant technology improvements, since they are based on the repetition of small units. This paper focuses on the self-adaptative features of a new reconfigurable architecture dedicated to the control from the application to the computation level. This reconfigurable device can itself adapt its resources to the application at run-time, and can exploit a high level of parallelism into an architecture called *RAMPASS*.

Index Terms *- dynamic reconfiguration, adaptative reconfigurable architecture, control parallelism*

1 Introduction

The silicon area of reconfigurable devices are filled with a large number of computing primitives, interconnected via a configurable network. The functionality of each element can be programmed as well as the interconnect pattern. These regular and modular structures are adapted to exploit future microelectronic technology improvements. In fact, semiconductor road maps [1] indicate that integration density of regular structures (like memories) increases faster than irregular ones (Tab. 1). In this introduction, existing reconfigurable architectures as well as solutions to control these structures, are first presented. This permits us to highlight the interests of our architecture dedicated to the control, which is then depicted in details.

A reconfigurable circuit can adapt its features, completely or partially, to applications during a process called reconfiguration. These reconfigurations are statically or dynamically managed by hardware mechanisms [2]. These architectures can efficiently perform hardware computations, while retaining much of

Year	1999	2001	2003	2005	2009	2012
Process (nm)	180	150	130	100	70	50
DRAM (bit/chip)	1,07 G	1,7 G	4,29 G	17,2 G	68,7 G	275 G
MPU (transistors/chip)	21 M	40 M	76 M	200 M	520 M	1,4 G

Table 1. Integration density for future VLSI devices

the flexibility of a software solution [3]. Their resources can be arranged to implement specific and heterogeneous applications. Three kinds of reconfiguration level can be distinguished :

- **gate level**: FPGA (Field Programmable Gate Array) are the most well-known and used gate-level reconfigurable architectures [4, 5]. These devices merge three kinds of resources: the first one is an interconnection network, the second one is a set of processing blocks (LUT, registers, etc.) and the third one regroups I/O blocks. The reconfiguration process consists in using the interconnection network to connect different reconfigurable processing elements. Furthermore, each LUT is configured to perform any logical operations on its inputs. These devices can exploit bit-level parallelism.
- **operator level**: the reconfiguration takes place at the interconnection and the operator levels (PipeRench[6], DREAM [7], MorphoSys [8], REMARC [9], etc.). The main difference concerns the reconfiguration granularity, which is at the word level. The use of coarse-grain reconfigurable operators provides significant savings in time and area for word-based applications. They preserve a high level of flexibility in spite of the limitations imposed by the use of coarse-grain operators for better performances, which do not allow bit-level parallelism.
- **functional level**: these architectures have been developed in order to implement intensive arithmetic computing applications (RaPiD [10], DART [11], Systolic Ring [12], etc.). These reconfigurable architectures are reconfigured in modifying the way their functional units are interconnected. The low reconfiguration data volume of these architectures makes it easier to implement dynamic reconfigurations and allows the definition of simple execution models.

These architectures can own different levels of physical granularity, and whatever the reconfiguration grain is, partial reconfigurations are possible, allowing the virtualization of their resources. Thus for instance, to increase performances (area, consumption, speed, etc.), an application based on arithmetic operators is optimally implemented on word-level reconfigurable architectures.

Besides, according to Amdahl's law [13], an application is always composed of regular and irregular processings. It is always possible to reduce and optimize the regular parts of an application in increasing the parallelism, but irregular code is irreductible. Moreover, it is difficult to map these irregular parts on reconfigurable architectures. Therefore, most reconfigurable systems need to be coupled with an external controller especially for irregular processing or dynamic context switching. Performances are directly dependent on the position and the level of this coupling. Presently, four possibilities can be exploited by designers:

- **microprocessor**: this solution is often chosen when reconfigurable units are used as coprocessors in a SoC (System on Chip) framework. The microprocessor can both execute its own processes (and irregular codes) and configure its reconfigurable resources (PACT XPP-/Leon [14], PipeRench [15], MorphoSys [8], DREAM [7], etc.). These systems can execute critical processings on the microprocessor, while other concurrent processes can be executed on reconfigurable units. However, in order to only configure and execute irregular code, this solution may be considered as too expensive in terms of area and energy consumption, and would most likely be the bottelneck due to off-chip communication overheads in synchronization and instruction bandwidth.
- **processor core**: this approach is completely different since the processor is mainly used as a reconfigurable unit controller. A processor is inserted near reconfigurable resources to configure them and to execute irregular processes. Performances can also be increased by exploiting the control parallelism thanks to tight coupling (Matrix [16], Chimaera [17], NAPA [18], etc.). Marginal improvements are often noticed compared to a general-purpose microprocessor but these solutions give an adapted answer for controlling reconfigurable devices.
- **microsequencer**: these control elements are only used to process irregular processing or to configure resources. They can be found in the RaPiD architecture, for instance, [10] as a smaller programmed control with a short instruction set. Furthermore, the GARP architecture uses a processor in order to only load and execute array configurations [19]. A microsequencer is an optimal solution in terms of area and speed. Its features do not allow itself to be considered as a coprocessor like the other solutions, but this approach is however best fitted for specifically controlling reconfigurable units. Nevertheless, control parallelisms can be exploited with difficulty.
- **FPGA**: this last solution consists in converting the control into a set of state machines, which could then be mapped to an FPGA. This approach can take advantage of traditional synthesis techniques for optimizing control. However, FPGA are not optimized for implementing FSM (Finite State Machines) because whole graphs of the application must be implemented even if non-deterministic processes occur. Indeed, these devices can hardly manage dynamic reconfigurations at the state-level.

Reconfigurable devices are often used with a processor for non-deterministic processes. To minimize control and configuration overheads, the best solution consists in tightly coupling a processor core with the reconfigurable architecture [20]. However, designing for such systems is similar to a HW/SW co-design problem. In addition, the use of reconfigurable devices can be better adapted to deep sub-microelectronic technological improvements. Nonetheless, the controller needs other physical implementation features rather than operators, and FPGA can not always be an optimal solution for computation. Indeed, the control handles small data and requires global communications to control all the processing elements, whereas the computation processes large data and uses local communications between operators.

To deal with control for reconfigurable architectures, we have developed the *RAMPASS* architecture (Reconfigurable and Advanced Multi-Processing Architecture for future Silicon Systems) [21]. It is composed of two reconfigurable resources. The first one is suitable for computation purposes but is not a topic of interest for this paper. The second part of our architecture is dedicated to control processes. It is a self-reconfigurable and asynchronous architecture, which supports SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data) and multi-threading processes.

This paper presents the mechanisms used to auto-adapt resource allocations to the application in the control part of *RAMPASS*. The paper is structured as follows: section 2 outlines a functional description of *RAMPASS*. Section 3 presents a detailed functional description of the part of *RAMPASS* dedicated to the control. This presentation focuses on some concepts presented in [21]. Then, section 4 depicts auto-adaptative reconfiguration mechanisms of this control part. Finally, section 5 presents the development flow, some results and deals with the SystemC model of our architecture.

2 Functional Description of RAMPASS

In this section, the global functionality of *RAMPASS* is described. It is composed of two main reconfigurable parts (Fig. 1):

- One dedicated to the control of applications (*RAC*: Reconfigurable Adapted to the Control);
- One dedicated to the computation (*RAO*: Reconfigurable Adapted to Operators).

Even if the *RAC* is a part of *RAMPASS*, it can be dissociated to be integrated with other different architectures with any computational grain. Each computation block can be either a general-purpose processor or a functional unit. The *RAC* is a generic control architecture and is the main interest of this paper. In this article, the *RAO* can be considered as a computational device adapted to the application, with a specific interface in order to communicate with the *RAC*. This interface must support instructions from the *RAC* and return one-bit flags according to its processes.

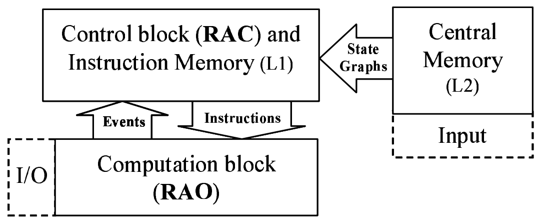
Control block (**RAC**) and Instruction Memory (L1)
State Graphs
Central Memory (L2)
Input
Events
Instructions
I/O
Computation block (**RAO**)

Fig. 1. Organization of RAMPASS

2.1 Overview

From a C description, any application can be translated as a *CDFG* (Control Data Flow Graph), which is a *CFG* (Control Flow Graph) with the instructions of the basic blocks expressed as a *DFG* (Data Flow Graph). Thus, their partition is easily conceivable [22, 23].

A *CFG* or a *State Graph* (*SG*) represents the control relationships between the set of basic blocks. Each basic block contains a set of deterministic instructions, called *actions*. Thus, every state in a SG is linked to an action. Besides, every arc in a SG either connects a state to a transition, or a transition to a state. A SG executes by firing transitions. When a transition fires, one token is removed from each input state of the transition and one token is added to each output state of the transition. These transistions determine the appropriate control edge to follow. On the other hand, a *DFG* represents the overall corresponding method compiled onto hardware.

Consequently, whatever the application is, it can be composed of two different parts (Fig. 2). The first one computes operations (*DFG*) and the second one schedules these executions on a limited amount of processing resources (*CFG*).

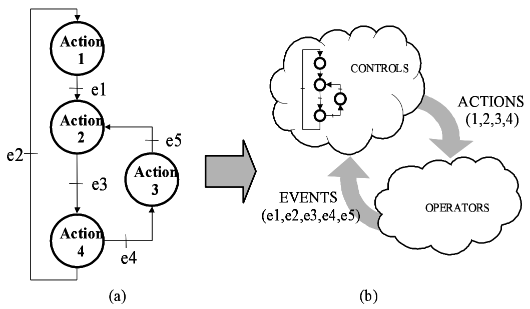

Fig. 2. Partitioning of an application (a) in Control/Computation (b)

The first block of our architecture can physically store any application described as a *CFG*. States drive the computation elements in the *RAO*, and events coming from the *RAO* validate transitions in the SG. Moreover, self-routing mechanisms have been introduced in the *RAC* block to simplify SG mapping. The *RAC* can auto-implement a SG according to its free resources. The *RAC* controls connections between cells and manages its resources. All these mechanisms will be discussed in future sections.

2.2 Mapping and running an application with RAMPASS

In this part, the configuration and the execution of an application in *RAMPASS* are described. Applications are stored in an external memory. As soon as the SG begins to be loaded in the *RAC*, its execution begins. In fact, the configuration and the execution are simultaneously performed. Contrary to microprocessor, this has the advantage of never blocking the execution of applications, since the following executed actions are always mapped in the *RAC*.

The reconfiguration of the *RAC* is self-managed and depends on the application progress. This concept is called auto-adaptative. The *RAC Net* has a limited number of cells, which must be dynamically used in order to map larger applications. Indeed, due to a lack of resources, whole SGs can not always be mapped in the *RAC*. Dynamic reconfiguration has been introduced to increase the virtual size of the architecture. In our approach, no pre-divided contexts are required. Sub-blocks implemented in the *RAC Net* are continuously updated without any user help. Figure 3 shows a sub-graph of a 7-state application implemented at run-time in a 3-cell *RAC* according to the position of the token.

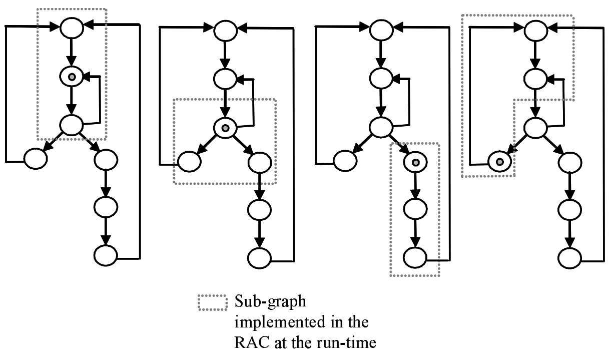

Fig. 3. Evolution of an implemented SG in the RAC Net

Each time a token is received in a cell of a SG implemented in the *RAC*, its associated instructions are sent to the *RAO*. When the *RAO* has finished its processes, it returns an event to the cell. This event corresponds to an edge in the SG mapped in the *RAC*. These transitions permit the propagation of tokens in SGs. Besides, each block has its synchronization mechanisms. In this globally asynchronous architecture, blocks are synchronized by 2-phase protocols [24].

It is possible to execute concurrently any parallel branches of a SG, or any independant SGs in the *RAC*. This ensures *SIMD*, *MIMD*, and multi-threading control parallelisms. Besides, semaphore and mutex can be directly mapped inside the *RAC* in order to manage shared resources or synchronization between SGs. Even if SGs are implemented cell by cell, their instantiations are concurrent.

3 Functional description of the control block: the RAC

As previously mentioned, the *RAC* is a reconfigurable block dedicated to the control of an application. It is composed of five units (Fig. 4). The *CPL* (*Configuration Protocol Layer*), the *CAM* (*Content Addressable Memory*) and the *LeafFinder* are used to configure the *RAC Net* and to load the *Instruction Memory.*

3.1 Overview

The *RAC Net* can support physical implementation of SGs. When a cell is configured in the *RAC Net*, its associated instructions are stored in the *Instruction Memory* as well as the address of its description in the *CAM*. Descriptions of cells are placed in a central memory and each description contains the instruction of the associated cell and the configuration of cells, which must be connected

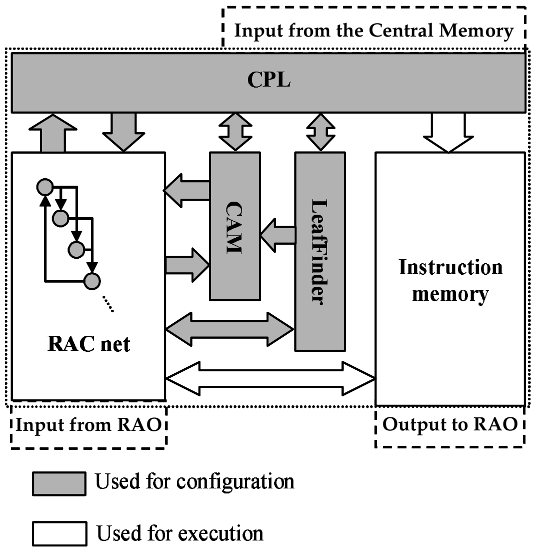

Fig. 4. The RAC block

(daughter cells). In order to extend SGs in the *RAC Net*, the last cells of SGs, which are called *leaf cells*, are identified in the *LeafFinder*. These cells allow the extension of SGs. When a leaf cell is detected, a signal is sent to the *CAM* and the description of this cell is read in the central memory. From this description, the daughter cells of this leaf cell are configured and links are established between the cells in the *RAC Net*. The *CAM* can also find a cell mapped in the *RAC Net* thanks to its address. This is necessary if loop kernels try to connect already mapped cells. Finally, the propagation of tokens through SGs, thanks to events from the *RAO*, schedule the execution of instructions stored in the *Instruction Memory*. In the next section, the details of each block are given.

3.2 Blocks description

RAC Net, this element is composed of cells and interconnect components. SGs are physically implemented thanks to these resources. One state of a SG is implemented by one cell. Each cell directly drives instructions, which are sent to the *RAO*. The *RAC Net* is dynamically reconfigurable. Its resources can be released or used at the run-time according to the execution of the application. Moreover, configuration and execution of SGs are fully concurrent. *RAC Net* owns primitives to ensure the auto-routing and the managing of its resources (cf §4.1). The *RAC Net* is composed of three one-hot asynchronous FSMs (5 ,8 and 2 states) to ensure the propagation of tokens, its dynamical destruction and the creation of connections. It represents about one thousand transistors in ST 0.18μm technology.

Instruction memory, the *Instruction Memory* contains the instructions, which are sent by the *RAC Net* to the *RAO* when tokens run through SGs. An instruction can eventually be either configurations or context addresses. As shown in figure 5, the split instruction bus allows the support of *EPIC* (Explicitly Parallel Instruction Computing) and the different kinds of parallelism introduced in the first section. Each column is reserved for a computation block in the *RAO*. For instance, the instructions A and B could be sent together to different computational blocks mapped in the *RAO* without creating conflicts, whereas the instruction C would be sent alone. A bit of selection is also used to minimize energy consumption by disabling unused blocks.

Furthermore, each line is separately driven by a state, e.g. each cell of the *RAC Net* is dedicated to the management of one line of this memory. This memory does not require address decoding since its access is directly done through its word lines. We call this kind of memory a *word-line memory.*

CPL, this unit manages SG implementation in the *RAC Net*. It sends all the useful information to connect cells, which can auto-route themselves. It drives either a new connection if the next state is not mapped in the *RAC net*, or a connection between two states already mapped. It also sends primitives to release resources when the *RAC Net* is full.

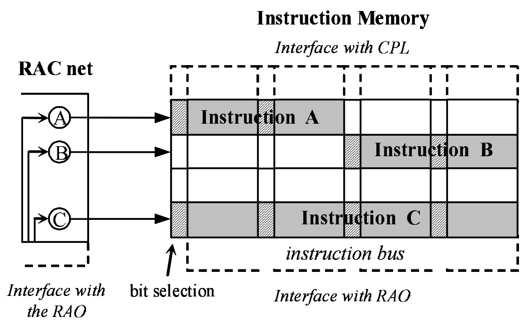

Fig. 5. Relation RAC Net/Instruction Memory

CAM, this memory links each cell of the *RAC Net* used to map a state of a SG, with its address in the external memory. Again, it can be driven directly through its word lines. It is used by the *CPL* to check if a cell is already mapped in the *RAC Net*. The *CAM* can select a cell in the *RAC Net* when its address is presented by the *CPL* at its input. Besides, the *CAM* contains the size of cell descriptions to optimize the bandwidth with the central memory.

LeafFinder, this word-line memory identifies all the leaf cells. Leaf cells are in a semi-mapped state which does not yet have an associated instruction. The research is done by a logic ring, which runs each time a leaf cell appears.

4 Auto-adaptative reconfiguration Control

The first part of this section deals with the creation of connections between cells and their configuration. A cell, which takes part in a SG, must be configured in a special state corresponding to its function in the SG. Finally, the second part focuses on the release of already used cells.

4.1 Graph creation and configuration

New connection To realize a new connection e.g. a connection with a free cell, the *CPL* sends a primitive called *connection*. This carries out automatically a connection between an existing cell (the source cell), which is driven by the *LeafFinder*, and a new cell (the target cell), which is a free cell chosen in the neighborhood of the source cell. Thus, each daughter in the neighborhood of the source cell are successively tested until a free cell is found. The *RAC Net* and its network can self-manage these connections. In fact, carrying out a connection consists of validating existing physical connections between both cells. Finally, the path between the two cells can be considered as auto-routed in the *RAC Net*.

Connection between two existing cells When the *RAC* finds the two cells, which must be connected, two primitives called *preparation* and *search* are successively sent by the *CPL* to the *RAC Net*. The first one initializes the research process and the second one executes it. The source cell is driven by the *LeafFinder* via the signal *start* and the target cell by the *CAM* via the signal *finish*. According to the application, the network of the RAC Net can be either fully or partially interconnected. Indeed, the interconnection network area is a function of the square of the number of cells in the RAC Net. Thus, a fully connected network should be used only in highly irregular computing application.

If the network is fully interconnected, the connection is simply done by the interconnect, which receives both the signals *start* and *finish*. On the other hand, if cells are partially interconnected, handshaking mechanisms allow the source cells to find the target. Two signals called *find* and *found* link each cells together (Fig. 6). On the reception of the signal *search*, the source cell sends a *find* signal to its daughters. The free cell receiving this signal sends it again to its daughters (this signal can be received only one time). So, the signal *find* spreads through free cells until it reaches the target cell. Then this cell sends back the signal *found* via the same path to the source cell. Finally, the path is validated and a hardware connection is established between the two cells. The intermediate and free cells, which take part in the connection, are in a special mode named *bypass*.

Configuration The dynamic management of cells is done by a signal called *accessibility*. This signal links every cell of a SG when a connection is done. Each cell owns an *Up Accessibility* (*UA*) (from its mother cells) and a *Down Accessibility* (*DA*) (distributed to its daughter cells). At the time of a new connection, a cell receives the *UA* from its mother cells and stores its configuration coming from the *CPL*. In the case of multiple convergences (details on SG topologies have been presented in [21]), it receives the *UA* as soon as the first connection is established. Then, a configured cell is ready to receive and to give a token. After its configuration, the cell transmits its *accessibility* to its daughters.

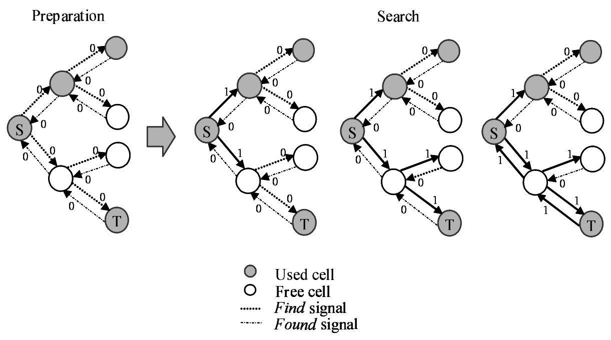

Fig. 6. Connection between the source cell (S) and its target cell (T)

When the connection has succeeded, the *RAC Net* notifies the *CPL*. Consequently, the *CPL* updates the *CAM* with the address of the new mapped state, the *LeafFinder* defines the new cell as a leaf cell, and the *Instruction Memory* stored the correct instructions.

When a connection fails, the *RAC Net* indicates an error to the *CPL*. The *CPL* deallocates resources in the *RAC Net* and searches the next leaf cell with the *LeafFinder*. These two operations are repeated until a connection succeeds. Release mechanisms are detailed in the next paragraph.

4.2 Graph release

A cell stops to deliver its *accessibility* when it no more receives an *UA* and does not own a token. When a cell loses its accessibility, all the daughter cells are successively free and can be used for other SG implementations. In order to prevent the release of frequently used cells, which may happen in loop kernels, a configuration signal called *stop point* can be used.

Due to resource limitations, a connection attempt may fail. For this reason, a complete error management system has been developed. It is composed of three primitives, which can release more or less cells. The appearing frequency of connection errors is evaluated by the *CPL*. When predefined thresholds are reached, adapted primitives are sent to the *RAC Net* to free unused resources. The first one is called *test acces*. It can free a cell in a *stop point* mode (Fig. 7). Every cell between two *stop point* cells are free. Indeed, a *stop point* cell is free on a rising edge of the *test acces* signal when it receives the *accessibility* from its mothers.

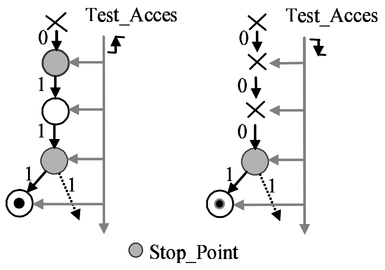

Fig. 7. Releasing of cells with test access

The second release primitive is named *reset stop point*. It can force the liberation of any *stop point* cells when they do not have any token. This mode keeps cells implied in the implementation of loop kernels and reduces the release. In some critical cases (when resources are very limited), it can become an idle state.

Finally, the last primitive called *reset idle state* guarantees no idle state in the *RAC*. This is done by freeing all the cells, which do not own a token. This solution is of course the more efficient but is very expensive in time and energy consumption. It must only be used in case of repeated desallocation errors.

No heuristics decide how many cells must be reclaimed or loaded. This is done automatically even if the desallocation is not optimal. That is why *stop point* cells must be adequately placed in SGs to limit releases.

Non-deterministic algorithms need to make decisions to follow their processes. This can be translated as OR divergences, e.g. events determine which branch will be followed by firing transitions. To prevent speculative construction and to configure too many unemployed cells, the construction of SGs is blocked until correct decisions are taken. This does not slow the execution of the application since the *RAC Net* contains always the next processes. Moreover, we consider that execution is slower than reconfiguration, and that an optimal computation time is about $3ns$. Indeed, we estimate the reconfiguration time of a cell equals to $7.5ns$ and the minimum time between two successive instructions for a fully interconnected network of $3ns + 1.5ns$, where $1.5ns$ is the interconnect propagation time for a 32-cell *RAC*.

5 Implementation and performance estimation

An architecture can not be exploited without a development flow. For this reason, a development flow is currently a major research concern of our laboratory (Fig. 8). From a description of the application in C-language, an intermediate representation can be obtained by a front-end like SUIF [22, 23]. Then, a parallelism exploration from the *CDFG* must be done to assign tasks to the multiple computing resources of the *RAO*. This parallelism exploration under constraints

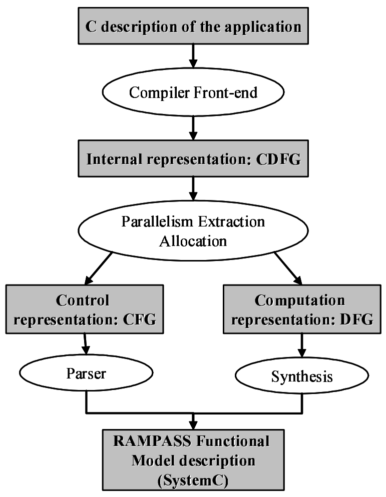

Fig. 8. RAMPASS Development Flow Graph

increases performances and minimizes the energy consumption and the memory bandwidth. The allocation of multiple resources in the *RAO* can also increase the level of parallelism. From this optimized *CDFG*, *DFGs* must be extracted in order to be executed on *RAO* resources. Each *DFG* is then translated into *RAO* configurations, thanks to behavioral synthesis scheme. This function is currently under development through the *OSGAR* project, which consists in designing a general-purpose synthesizer for any reconfigurable architectures. This *RNTL* project under the ward of the French research ministry, associates TNI-Valiosys, the Occidental Brittany University and the R2D2 team of the IRISA. On the other hand, a parser used to translate a *CFG* into the *RAMPASS* description language, has been successfully developed.

Besides, a functional model of the *RAC* block has been designed with SystemC. Our functional-level description of the *RAC* is a CABA (Cycle Accurate and Bit Accurate) hardware model. It permits the change of the size and the features of the *RAC Net* and allows the evaluation of its energy consumption. The characteristics of this description language easily allows hardware descriptions, it has the flexibility of the C++ language and brings all the primitives for the modelization of hardware architectures [25, 26].

A lot of different programming structures have been implemented in the *RAC* block, e.g. exclusion mechanisms, AND convergence and divergence, synchronizations between separated graphs, etc. Moreover, an application of video

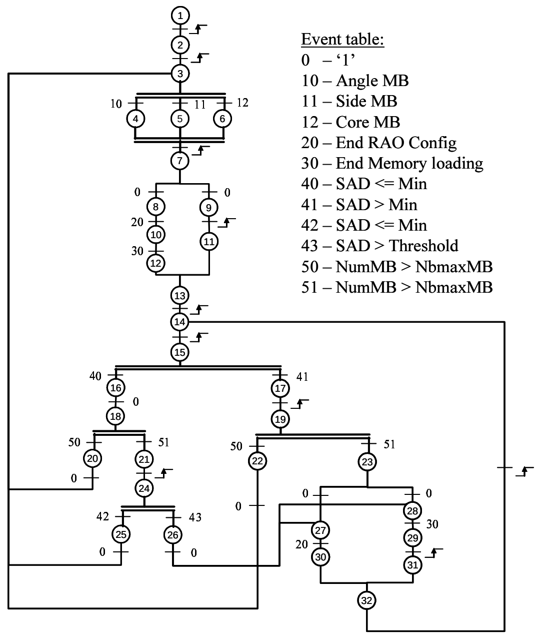

Fig. 9. Motion estimation graph

processing (spinal search algorithm for motion estimation [27]) has been mapped (Fig. 9). The latency overhead is insignificant without reconfiguration when the *RAC* owns 32 cells, or with a 15-cell *RAC* when the whole main loop kernel can be implemented (0.01%), even if we cannot predict reconfigurations. Finally with a 7-cell *RAC* (the minimal required for this application), the overhead raises only 10% in spite of multiple reconfigurations, since the implementation of the SG must be continuously updated.

Besides, hardware simulations have shown the benefits of release primitives. Indeed, the more cells are released, the more the energy consumption increases since they will have to be re-generated, especially in case of loops. Simulations have shown that these releases are done only when necessary.

Some SG structures implemented in the *RAC Net* need an imperative number of cells. This constraints the minimal number of cells to prevent dead-locks. For instance, a multiple AND divergence has to be entirely mapped before the token is transmitted. Consequently, an 8-state AND divergence needs at least nine cells to work. Dynamic reconfiguration ensures the progress of SGs but can not prevent dead-locks if complex structures need more cells than available inside the *RAC Net*. On the contrary, the user can map a linear SG of thousands of cells with only two free cells.

6 Conclusion and future work

New paradigm of dynamically self-reconfigurable architecture has been proposed in this paper. The part depicted is dedicated to the control and can physically implement control graphs of applications. This architecture brings a novel approach for controlling reconfigurable resources. It can answer future technology improvements, allow a high level of parallelism and keep a constant execution flow, even for non-predictible processing.

Our hardware simulation model has successfully validated static and dynamic reconfiguration paradigms. According to these results, further works will be performed. To evaluate performances of RAMPASS, a synthesized model and a prototype of the *RAC* block is currently designed in a ST 0.18μm technology.

Moreover, the coupling between the *RAC* and other reconfigurable architectures (DART, Systolic Ring, etc.) will be studied. The aim of these further collaborations consists in demonstrating the high aptitudes of the *RAC* to adapt itself to different computation architectures.

7 Acknowledgements

We thank Dominique Lavenier (IRISA, France), Laurent Letellier and Raphaël David (CEA, France) for helpful discussions and comments on this work.

References

1. Semiconductor Industry Association. International Technology Roadmap for Semiconductors. Technical report, 2003.

2. K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.
3. R. Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *IEEE Design Automation and Test in Europe (DATE)*, Munich, Germany, March 2001.
4. Xilinx, http://www.xilinx.com.
5. Altera, http://www.altera.com.
6. S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer: Innovative Technology for Computer Profesionals*, 33(4):70–77, April 2000.
7. J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk. Fast Communication Mechanisms in Coarse-grained Dynamically Reconfigurable Array Architectures. In *Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE)*, Las Vegas, USA, June 2000.
8. H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. on Computers*, Vol.49, No.5:465–481, May 2000.
9. T. Miyamori and K. Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *ACM/SIGDA Field Programmable Gate Array (FPGA)*, Monterey, USA, February 1998.
10. D. Cronquist. Architecture Design of Reconfigurable Pipelined Datapaths. In *Advanced Research in VLSI (ARVLSI)*, Atlanta, USA, March 1999.
11. R. David, S. Pillement, and O. Sentieys. *Low-Power Electronics Design*, chapter 20: Low-Power Reconfigurable Processors. CRC press edited by C. Piguet, April 2004.
12. G. Sassateli, L. Torres, P. Benoit, T. Gil, G. Cambon, and J. Galy. Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP applications. In *IEEE Design Automation and Test in Europe (DATE)*, Paris, France, March 2002.
13. G.M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings vol.30*, Atlantic City, USA, April 1967.
14. J. Becker and M. Vorbach. Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC). In *IEEE Computer Society Annual Workshop on VLSI (WVLSI)*, Florida, USA, February 2003.
15. Y. Chou, P. Pillai, H. Schmit, and J.P. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Symposium on Microarchitecture (MICRO-33)*, Monterey, USA, December 2000.
16. B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *Design Automation and Test in Europe (DATE)*, Paris, France, February 2004.
17. Z. Ye, P. Banerjee, S. Hauck, and A. Moshovos. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled RFU. In *the 27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, June 2000.
18. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, USA, April 1998.

19. J.R. Hauser and J. Wawrzynek. GARP: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, USA, April 1997.
20. D. Rizzo and O. Colavin. A Video Compression Case Study on a Reconfigurable VLIW Architecture. In *Design Automation and Test in Europe (DATE)*, Paris, France, March 2002.
21. S. Chevobbe, N. Ventroux, F. Blanc, and T. Collette. RAMPASS: Reconfigurable and Advanced Multi-Processing Architecture for future Silicon System. In *3rd International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, July 2003.
22. G. Aigner, A. Diwan, D.L. Heine, M.S. Lam, D.L. Moore, B.R. Murphy, and C. Sapuntzakis. The Basic SUIF Programming Guide. Technical report, Computer Systems Laboratory, Stanford University, USA, August 2000.
23. M.D. Smith and G. Holloway. An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization. Technical report, Division of Engineering and Applied Sciences, Harvard University, USA, July 2002.
24. I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.
25. J. Gerlach and W. Rosenstiel. System level design using the SystemC modeling platform. In *the 3rd Workshop on System Design Automation (SDA)*, Rathen, Germany, 2000.
26. S. Swan. An Introduction to System Level Modeling in SystemC 2.0. Technical report, Cadence Design Systems, Inc., May 2001.
27. T. Zahariadis and D. Kalivas. A Spiral Search Algorithm for Fast Estimation of Block Motion Vectors. In *the 8th European Signal Processing Conference (EUSIPCO)*, Trieste, Italy, September 1996.