Control unit for parallel embedded system

Stéphane Chevobbe, Raphael David, Frederic Blanc, Thierry Collette
CEA-List DRT/DTSI/SARC/LCEI
F-91191 Gif sur Yvette
FRANCE
{stephane.chevobbe,raphael.david,frederic.fblanc,thierry.collette}@cea.fr

Olivier Sentieys
IRISA - ENSSAT
6 rue de Kérampont
22305 Lannion Cedex
olivier.sentieys@enssat.fr

ABSTRACT
New integration methodologies as IP reuse have become more and more popular since few years. These methodologies represent an opportunity to reduce the gap between the integration capacities and the ability of the designers to develop complex systems. SOC (System On Chip), that are composed of different heterogeneous cores, have taken benefit of these methodologies. Recent SOCs are usually Globally Asynchronous and Locally Synchronous and exploit a lot of parallelism. Up to now, research efforts have mainly been focused on the definition of new communication and processing primitives. Unfortunately, control mechanisms have not evolved as fast as the rest of the system. Besides, these devices are usually organized around a microprocessor supporting an operating system managing the execution of the different tasks on the processing primitives and their communications. We propose in this paper a hardware solution to manage tasks and communication in such parallel systems. This controller focus on the implementation of Petri nets and has the property to be reconfigurable and to self-manage its configurations. Physical implementation of this component has been done in 0.13µm technology.

Keywords
SOC, parallel controller, Petri-net, asynchronous, GALS

1. Introduction
In one hand, to take benefits from technology improvements while trying to reduce more and more design costs, recent design methodologies are based on IP reuse. In the other hand, to achieve more and more computing power new execution models are proposed. Here are introduced the main execution models proposed in the literature.

Commercial products based on the SOC paradigm are often dedicated to specific domains. Those designs are usually based on a central CPU, which manages hardwired accelerators executing time-consuming tasks. For example, the platforms Nomadik architecture from STMicroelectronics [1] and the Nexperia [2] from Philips are dedicated to multimedia applications. They are composed of a central CPU which manages with I/O controllers and hardware accelerators optimize for audio or video. In these platforms the control is assumed by an OS running on the host processor. The main drawbacks of this kind of solutions are both the lack of scalability and their poor ability to execute parallel applications. Due to the CPU, the execution model of such SOC is too sequential to achieve high level of parallelism, and because of centralized control, these architectures can not be extended keeping the same organization.

On the contrary, dataflow architectures propose to use a spatial distribution of tasks on the chip. IPs are connected by a network on chip and synchronizations between tasks are performed by the data [3]. As an example the RAPID architecture [4] is composed of ALU and memories elements connected by a reconfigurable network. These approaches are very efficient for regular applications, whose control flows can be statically predicted, but are not suitable for irregular ones.

Hybrid architectures try to solve drawbacks of the two previous solutions by mixing them. Hybrid SOCs are composed of two kinds of resource. The first one is in charge of the task level control. The second one is composed of processing elements that can be eventually heterogeneous (specific data-path or reconfigurable structures). As example, DART is a reconfigurable coarse-grained architecture in which the operators and the data-path are dynamically reconfigurable [5]. PACT proposes the XPP [6] architecture which is a dynamically reconfigurable data-path composed of DSP, ALU and memory elements. Even if these solutions offer some real advances in embedded computing, they still fail to cope with dynamic control flows which needs tight coupling between control and computations and reactive solutions.

As we briefly show upper, the management of computation and exploitation of the potential parallelism of the application is a key for the system performance. The kind and the amount of parallelism may however significantly vary between two applications, and even between two implementation of a same application. Parallelism properties have thus to be extracted from the applications.

It can be done by extracting and analyzing data-dependencies between these applications [7]. However, once extracted, the parallelism will also be limited by the computation model of the hardware structure. For example, SIMD architecture will only efficiently exploit similar independent tasks because of its control structure. Furthermore, hardware structures may add extra constraints to the system, such as the sharing of memories which imply the introduction of synchronization primitives in the application control flow.

The aim of the study is to propose a control solution that does not add those extra constraints. The paper is organized as follow. The next section presents the control requirements at system level and a state of the art of hardware implementing Petri Nets. The third section introduces the RAMPASS platform and the fourth describes in detail our controller named RAC. The fifth section presents performances and implementation results of the architecture.

2. System control requirement

2.1 Definition of control functionalities

Control is defined as a process able to make decisions regarding a set of events and a history of previous taken decisions. Four kinds of control structures can be used to describe the control flow of an application:

1 Free Choice: a set of actions is possible depending on the forthcoming events. In a software point of view this is equivalent to a "if…then…else…" structure.

2 Parallel divergence: This control structure creates branches in the decision tree. It is equivalent to a ***fork*** instruction used in parallel language.

3 Parallel Convergence: This control structure joins decision branches. It is equivalent to a ***join*** instruction used in parallel language.

4 Mutual exclusion: This structure permits to manage synchronizations due to any shared resources.

2.2 Control system and Petri net

Finite state machines have been intensively used to describe the control flow of application in specific integrated circuit, it is not well suited for concurrency description. This model imposes to describe each states of the system, which can be enormous in case of concurrent behavior. Petri nets have been proposed as an alternative modeling formalism to exploit the advantages that they offer over automata models [8]. Petri net models are generally more compact and more powerful than automata models. They are a powerful formalism for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel and non deterministic [9].

A *Petri Net* is a particular kind of directed graph with an initial state called the initial marking. The underlying graph of a Petri net is a directed, weighted, bipartite graph consisting of two kinds of nodes, called places and transitions. Edges connect either a place to a transition and a transition to a place. A change in the net marking corresponds to the execution of actions (known as transition firing or occurrence). These changes occur according to the following basic rules:

(a) a transition is enabled if every of its input places has at least one token;

(b) any enabled transition can occurs. Their firing removes a token from each of the corresponding input places and insert a new token in all their output places;

(c) enabled transitions can occur concurrently as long as they are independent, i.e., as long as they use different tokens.

Different subclass of Petri net can be described according to their decision and modeling power. SMs (State Machine) admit no synchronization, MGs (Marked Graphs) admit no conflict, FCs (Free Choice) allows asymmetric confusion but disallow symmetric confusion which are admitted by PNs (Pure Net) [10].

On a control point of view, the computational units in the system need only an untimed model, in this sense that there is only a sequence of actions that have to be completed. Some actions must occur before others to respect data precedence constraints and causality, whereas others can occur in parallel. No timing considerations are needed for the controller to guarantee this action scheduling.

2.3 Related work

The use of controllers based on Petri net is very large. The first architectures were dedicated to the control of asynchronous concurrent systems. A lot of others applications have also been found from that time, as industrial systems control or used in synthesis methodology of asynchronous design [10].

First published hardware implementations of Petri Nets controller were developed by S. S. Patil in 1972 [11]. They propose this kind of controller because of the ease of describing concurrent behavior with Petri Nets. These implementations were based on the use of flip-flops to substitute each individual place and transition.

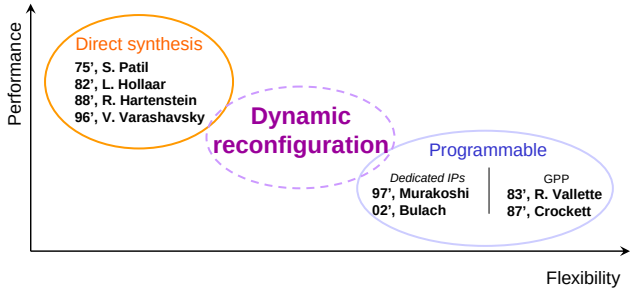

Figure 1: Architectural space of controller based on Petri Net

The architectures implementing Petri Net can be classified according to the execution model to compute the PN and the target architecture on which is implemented the PN. Globally, the space of these architectures can be divided in two fields as shown in figure 1. The first one is related to the implementation of a particular Petri Net, while the other support the modification of the implemented Petri net after the component fabrication. In each field, several architectural targets are available to implement the Petri Net. Hardwired as well as programmable processors controllers can be found in the literature.

2.3.1 Direct implementation

When Petri nets are directly synthesis in hardware with glue logic and flip-flops it is called direct implementation. A configurable architecture based on the *Kolte Array* is proposed in 1988 by R. W. Hartenstein and al. [12]. Thus, the architecture is a little more flexible than dedicated one. The main drawback is the interconnect network between the places and the transitions. It is a crossbar to allow connection between each place and transition but it is under used because of static configuration. This kind of architecture is used to implement safe Petri nets, which are a subclass of Petri nets with no more than one token at a place and where no place is both the input and the output of the same transition.

A dataflow machine based on implementation of Petri net is proposed by J. P. David [13]. Different nodes are used to implement an application. The methodology is limited by the interconnection. Indeed, the size of the bus that connects the nodes of the net is equal to the size of the data.

2.3.2 Programmed implementation

Petri net implementations based on commercial platforms are based on the use of general-purpose processors which do not inherently behave in an event-driven manner. S. Bulach proposes a dedicated programmable controller based on the execution of PN [14]. Starting from the net structures supporting concurrent processes and the inclusion of I/O signals in the firing conditions,

an encoding scheme for the storage of the net structure in an EPROM memory is generated. The cyclic net execution algorithm consists of transition enabling checks and firings. New output signals are set after all enabled transitions of the current marking have been fired. Although the execution model of the controller is based on PN, the computation of the next marking is sequential.

Performances of the architectures implementing Petri Net are a limitation to use it as a control unit of systems on chip. We thus propose an architecture which takes advantage of the flexibility of programmable solutions without reducing performance (cf. figure 1). We propose in the next section a new computation platform, named RAMPASS (Reconfigurable And Advanced Multi-Processing Architecture for future Silicon Systems), whose control unit is based on an efficient implementation of Petri nets.

3. RAMPASS platform

RAMPASS is divided in three parts [15], depicted in figure 2. An application memory stores the complete description of the application. The active control part (RAC: Reconfigurable Architecture for the Control) manages the tasks execution on all the resources of the computing part (RAO : Reconfigurable Architecture for the Operators). The computing part is composed of processing elements that receive commands from the control part. Processing elements can be programmable processors, reconfigurable operators or hardwired accelerators.

This platform is event driven and is able to manage synchronous, asynchronous, as well as hybrid Globally Asynchronous Locally Synchronous systems. Because of the modular structure of SOC, this control part has to be able to manage concurrently several independent control flows to manage the different computing primitives independently.

To be implemented in the RAC, the application has to be splitted into tasks and described as a safe labeled Petri net. Places describe local states of the application execution and transitions are associated to a command/event couple. The command is sent to the computing part to start the execution of a computation. The events point a specific transition of the application that evaluates the state of the execution. This description formalism will be detailed in section 5.2.

The RAC executes active windows of the whole application described in the *application memory*. These windows depict the parts of the application containing at least one active place in the Petri net describing the application. It is continuously updated by internal building and release mechanisms. The set of tokens propagated in the window is sufficient to describe the global state of the system. Each active token in the window will trigger the sending of a command to the computing part. Next, the computing part executes the commands as soon as they are received, and sent back event at the end of the executions. Finally, according to these events, all the tokens move from the input places of the transitions to their output places. As the events are sent independently by computing resources of the RAO, several tokens can evolve in parallel.

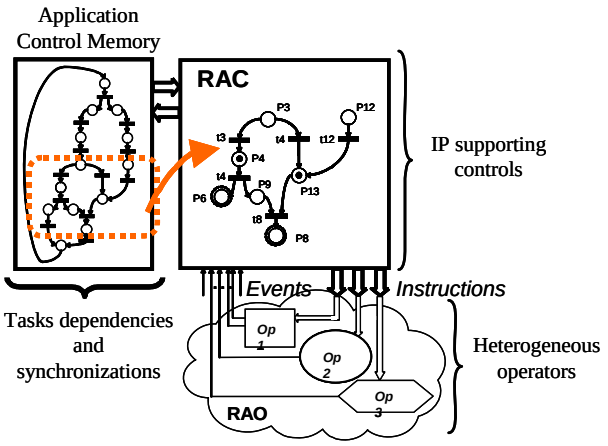

Figure 2: RAMPASS overview

There are thus two interfaces between the control part and the computing part. The first one (*command interface*) is used to send the commands to the computing resources and the second one (*event interface*) allows to send events from the computing part to the control part. The *command interface* implements a simple two-phase handshaking protocol.

4. RAC description (*Reconfigurable Adapted to Control*)

The control part of RAMPASS is an architecture dedicated to the implementation of PN. We discus at first in this section its architectural principles. In subsection 4.2 we will detail some prototyping issues of this solution. Finally section 4.3 will clarify the application description formalism.

4.1 Architectural description

The RAC implements directly PN on a dynamically auto-adaptable architecture. It self-manages its hardware resources, through the realization of three basic functions:

- Execution: the RAC fires transitions when all their input places own a token and when its associated condition is high. The execution is in charge of moving the tokens from input place of transitions to their output places. Finally, it sends the associated commands to transitions.
- Building: The RAC is able to identify the branches of the active windows that have to be extended. It is also able to place and route new cells on its network of cells to implement the new active windows.
- Release: This action is needed to free resources on the RAC network of cells. Basically, cells without token in their input branches are released slowly but continuously. When there is an urgent need of resources, the RAC is also able to release concurrently all the cells of its input branch.

All these three functions are done concurrently to optimize the use of the hardware. This makes the architecture dynamically reconfigurable. All the decisions concerning the allocation and the routing of the PN build in the hardware are done by the architecture itself. Thanks to the self managing of the building and release actions, the description of the application can be shortened. Indeed no place and route information have to be added to the functional description of the PN to implement it in

the RAC. These properties enable new tradeoff between flexibility (programmable processor) and performance (direct implementation of PN).

To illustrate the execution model of the RAC a simple example can be used. Let consider at first a network of cells totally free. A boot sequence configures a *boot cell* from which the whole net will be built. This boot cell points the initial marking of the PN in the application memory. The architecture has next to select enough free cells in the hardware structure to begin the implementation of the PN. Each new cell is then selected to develop every branch of the PN starting from the boot cell. These new cells are named *leaf cells.*

When a *leaf cell* is selected to build a new cell, its description is loaded from the application memory. For each cell connected from the selected leaf, two cases can occur. Indeed the cell can be already mapped in the cell network. In that case, the connection is directly performed. On the contrary, if the cell is not mapped in the network, a free cell has to be selected before to perform the connection. Next the graph continues to be developed, by searching new leaf cells. When all the cells are used, reset processes are executed to release cells. In this way, the PN is continuously extended.

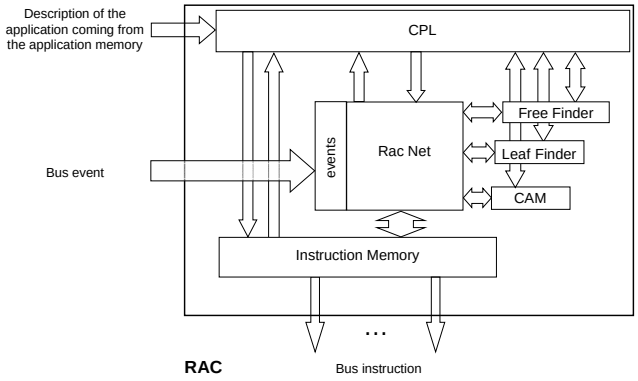

Figure 3: RAC overview

The RAC is divided into six basic blocs as shown in figure 3. The *RAC net* and the *instruction memory* are involved in the execution of the PNs. The others modules (*Configuration Protocol Layer*, *Freefinder*, *Leaffinder*, *Content Addressable Memory*) are only involved in the management of the resources.

4.1.1 CPL

The *Control Protocol Layer* manages at high level the implementation of the PN on the hardware. It triggers the different steps of the building and release of the PN in the hardware. It is not involved in the execution of the PN.

4.1.2 CAM

Pointers on mapped cells are loaded in a Content Addressable Memory (CAM). These pointers are used to find and locate nodes that are already mapped in the network. The CAM selects the node whose pointer is set by the CPL. If it is not present in the CAM the cell is not mapped and the CPL as to load it in the network. This CAM has been designed with standard registers and XOR gates.

4.1.3 Freefinder & Leaffinder

These two elements locate free cells and leaf cells in the network. They are implemented thanks to asynchronous round robins. Each free cell and leaf cell set up flags that are sent to these asynchronous round robins to select one cell at a time. These elements can be seen as asynchronous arbiter.

4.1.4 Rac net

The RAC net is composed of a set of cells, representing places and transition of the PN communicating through a reconfigurable network. In order to support the three basic functionalities explained previously, each cell is composed of three processes. Each of them is implemented as an asynchronous one-hot FSM. Additional synchronizations between FSMs guarantee the coherency of the system.

The execution part implementing nets has to allow multiple connections between places and transitions (to implement convergence and divergence in the PN). Consequently, the events coming from the operative part have to be distributed to all the cells. The links between the event and the mapped places are done during the configuration of the cell.

The interconnection network allows connections between every cell. Each cell can be connected with any other one through a hazard free full-connected network. The mapping strategy is thus very simple. The connections between two cells are performed by selecting of the source and the destination cells. A simple crossing enables to find the connection point between two cells. When two cells are connected together two kinds of information goes through the network:

1. Tokens: they are implemented by asynchronous hand-shaking between cells. Tokens are thus modeled as synchronization mechanisms between two cells.
2. Accessibility flags: this information permits to identify cells that can be released. Indeed a cell that is not connected to any other could not receive any tokens and is thus useless.

4.1.5 Instruction Memory

The instruction memory sends tasks to the operative part. This memory is particular for several aspects. First, the cells directly command the line of the memory. This memory has thus no address decoder. Secondly, the memory is splitted in several flows which are independent and can be accessed concurrently. A cell can thus command independently either one or several blocks of a memory line according to its configuration. Consequently, as several blocks of the memory can be activated in parallel, the controller can manage several tasks in parallel.

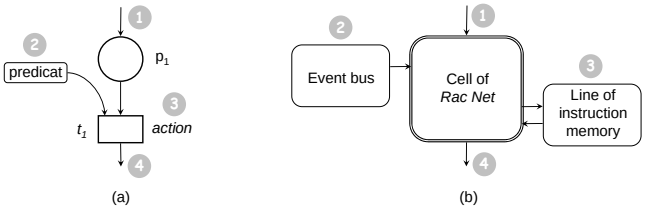

Figure 4: Equivalence between place and transition of PN formalism and hardware cell of the RAC.

4.2 Implementation

A prototype of the RAC has been designed using a 0.13µm design kit provide by STMicroelectronics (HCMOS9). Logical synthesis flow from Synopsys has been used for front-end while backend has been performed with Cadence Soc Encounter.

A first model has been developed in SystemC to provide fast and accurate simulations. A set of PNs testbenches has been used to validate the architecture and provide a good functional coverage (up to 90%). Each implemented graph validates basic functionalities as parallelism support, tasks synchronization or dynamic reconfiguration.

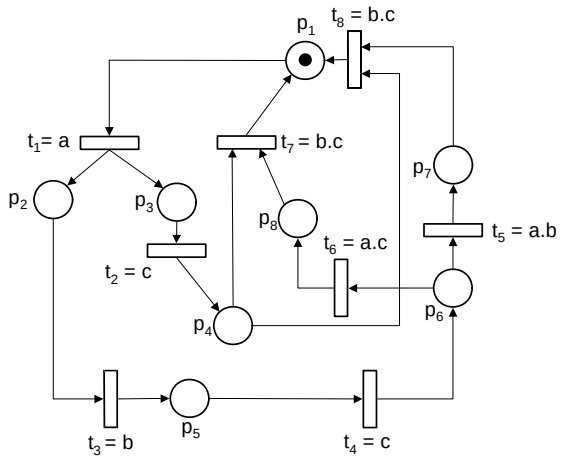

Figure 5: Interpreted Petri Net with {a;b;c} as event

4.3 Application description

The formalism used to describe a safe *labeled Petri net*, executable by the RAC, respects the theory of Petri. As shown in the figure 4, each cell of the RAC corresponds to a couple place/input transitions. The hardware and the function of routage are simplified since the network does not have to connect heterogeneous resources.

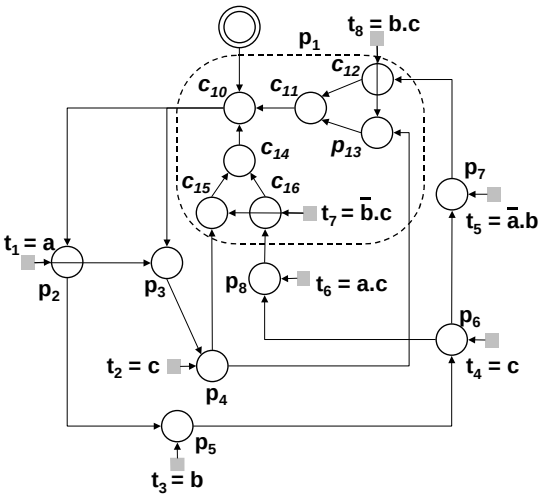

Figure 6: Translation of the PN figure 4 in the formalism understandable by the RAC

As said before, the whole graph is described in a static memory. Places of the PN are identified with a memory address and an offset corresponding to the size of the description. Input events, tasks, modes, and following cells addresses are contained in the description of every cell. Each cell is associated to input ports to receive the incoming events and tasks acknowledgement, and to an output port control the associated task.

These cells allow the implementation of pure nets. The figures 5 and 6 respectively show a simple Petri net and the associated implemented graph on the RAC. Except places p_1, which needs cells $\{c_{10}; c_{11}; c_{12}; c_{13}; c_{14}; c_{15}; c_{16}\}$ to be mapped, all the others places just need one cell of the hardware structure. This transformation thus infers only a small overhead between on, the PN modeling the application.

5. Results

The results presented below come from the analysis of the logical synthesis and the post place-and-route models. The gate-level netlist is used to study the scalability of the architecture while the post place-and-route model is used to exhibit accurate area and performance results. This section is divided in three parts. The first one presents some implementations results. The second discusses the performances of the architecture and the third subsection analyzes the implementation of a MPEG4-AVC application into the RAC.

5.1 Layout analysis

Because of a limited amount of pins available for our prototype, it is composed of 16 cells and provides 6 8-bit independent instruction streams. The area of the prototype is 0.81 mm^2 of which only 0.2 mm^2 are needed for the RAC. The figure 7 shows this layout.

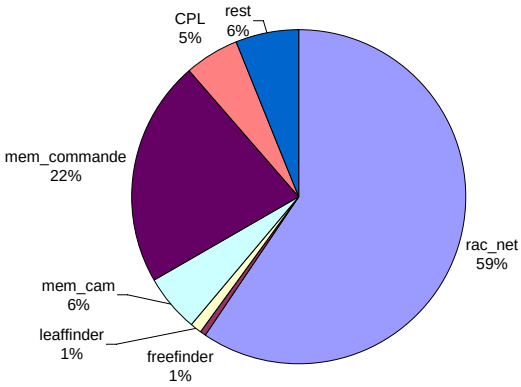

Figure 8: Functional distribution of the RAC in area

The area distribution in the chip (Fig. 8) is relatively well balanced. Indeed, the execution part of the design fills 81 % of the layout, dynamic management of the cells network and debug resources of the circuit fill the rest. The interconnect network occupies 59 % of the area for a 16-cell RAC. For a 60-cell RAC, this interconnect will grows up to 79 % of the area. Its size evolving in $O(N^2)$, where N is the amount of cell,, it limits the scalability of this architecture.

The modules of the resource management part are well balanced and their areas are homogeneous. Their contributions fall under 5% of the area for a RAC with 60 cells. The RAC had been synthesized for different number of cells, ranging from 8 to 60. The minimum area obtained is 0.6 mm^2 (mainly devoted to the application memory) and the maximum is 2.9 mm^2.

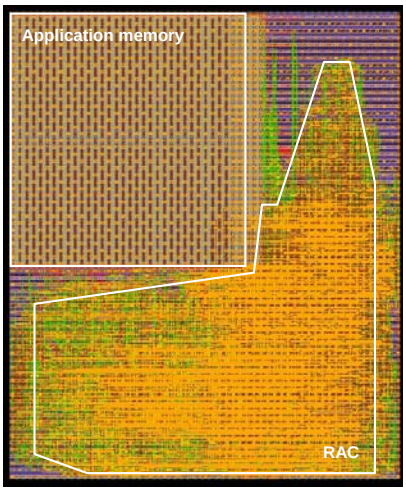

Figure 7: Layout of the RAC and the Application Memory

5.2 Performances

The two main features discussed in this section are the response time of the RAC (in ns) and the output bandwidth of the *instruction memory* (in Million Commands Per Second: MCPS). The response time depicts the time between the arrival of an event at the input of the RAC and the triggering of the corresponding output in the *instruction memory*.

These two parameters depend on the execution mode, i.e. static or dynamic. The static mode depicts the execution of a PN completely mapped on the RAC, while the dynamic one concerns the executions of PNs bigger than the RAC capabilities, that need to build and to release new cells during the execution. The static mode limits strongly the complexity of the application, which can be mapped on the architecture, but it gives the maximum performance. On the contrary, the dynamic mode allows mapping very complex application, but is associated to a bigger timing overhead.

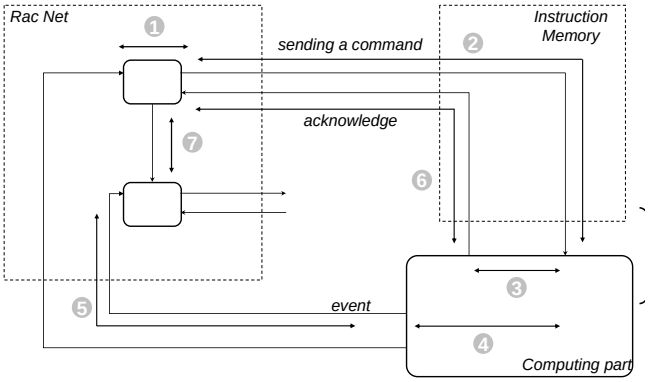

Figure 9: Timing elements in the execution cycle : 1 time between the arrival of an event and the sending of a command ; 2 time between the sending of a command and its reception by the computing part ; 3 response time of the computing part ; 4 execution time of the task ; 5 propagation time of the event ; 6 propagation time of the acknowledge ; 7 propagation time of the tokens between two cells.

In the static mode, the response time is independent of the graph mapped into the RAC. This time is due to a sequence of actions to be done in the RAC (1, 2 and 5 in the figure 9) to propagate signals between events, cells and commands. Even in the worth conditions this time is lower than 10ns.

The total bandwidth of the *instruction memory* is directly proportional to the amount of command flows that are configured. So in calculating the bandwidth of one flow the total bandwidth can be determined. The bandwidth of one flow is inversely proportional to the sum of the times 1, 2, 3, 6 and 7. This bandwidth for one flow is 48 MCPS. It can go up to 288 MCPS when the 6 command flows are used.

In dynamic mode other timing constraints, mainly due to resources management, have to be taken into account. This timing highly depends on the graph mapped into the RAC. The evolution of the active windows within the hardware is very hard to predict. It is thus not possible to define all the contributions involved in the response time and in the bandwidth of the *instruction memory* as for the static mode. Nevertheless, the building of a graph is highly sequential and is done by the CPL. It takes 65 ns (13 cycles at 200 MHz) to configure every new cell. The CPL could have been synthesized at 400 MHz but due to limitations on the input clock pin of the prototype, a slower version has been implemented.

Lots of simulation involving dynamic graphs had shown that the bandwidth of the *instruction memory* is ranging from 8 MCPS to 15 MCPS in average. Although the bandwidth can not be computed for a given application, the maximum is limited by the building of a cell. Indeed, the upper limit is equal to:

$$Max_{MCPS} = \frac{1}{T_{build}}$$, where T_{build} is the time of build of one cell.

The response time can however be more precisely defined. Several cases can occur according to the execution time. First, if the execution on the RAO is low enough to hide the dynamic behaviour, the response time will be identical to that of the static mode. Conversely, if the execution time is in the same range than the building time, the response time will correspond to that of the building (70 ns).

The RAC performances have been studied with several numbers of cells. The response time is relatively independent of it. The upper limit of the bandwidth is however clearly dependent of this number. The figure 10 shows both the evolution of the maximum static bandwidth for a flow versus the number of cells into the RAC and the maximum bandwidth for the instruction memory.

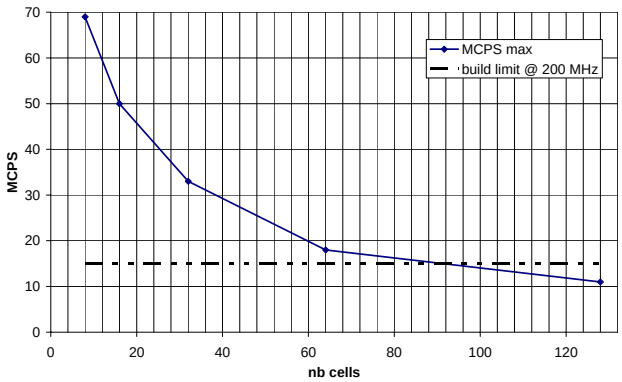

Figure 10: MCPS versus number of cells in the RAC

It appears in figure 10 that the MCPS highly depends on the number of cells. For a 8-cell RAC, the maximum frequency for a flow is 70 MCPS. For a 128-cell RAC it falls to 11 MCPS. This is due to the contribution of the interconnect network which become prohibitive (time 7 on figure 9). Secondly, the crossing between the horizontal lines and the MCPS curve points the number of cells where the building time and the maximum bandwidth is the best balanced. According to the prototype results this 100 cells seems to be a good trade-off.

The performance analysis reveals two drawbacks. First, the interconnect network is a limitation in term of area and performance. To improve the performances of the architecture, a better communication media have to be studied. Secondly, the management of resource is too sequential. It has been clearly shown that the dynamic management has a strong impact on the performance. Parallelizing the building process will thus significantly improve the performance of the controller.

5.3 Case study

As an example we have implemented the motion estimation of the MPEG-4 AVC encoder using the RAMPASS approach. The PN model of the application required 60 states with 6 branches, several synchronizations and parallel execution. The partionning of the application has been studied in [16] and has been done according to the analysis of the control flow and the data flow.

The figure 11 shows some simulation results for a 16-cell RAC. It shows the execution overhead due to RAC according to the average task length. For a video in QCIF format the partionning leads to tasks whose average execution time is 4 µs. The RAC overhead is that case is very low. For tasks granularity around 1 µs the overhead grows up to 5%. Finally for shorter tasks like that extracted to handle VGA video (300 ns) the RAC overhead become significant (12,1%).

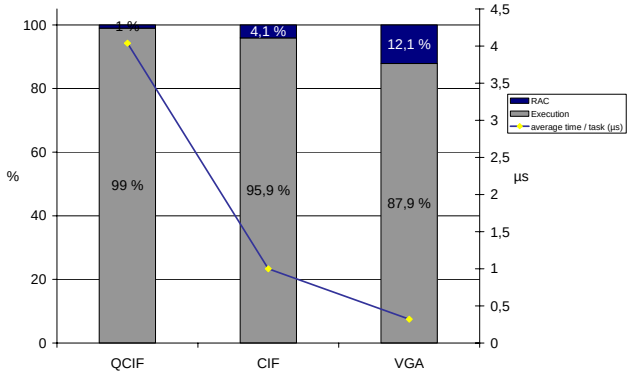

Figure 11: Overhead due to the RAC versus granularity of task.

In this case study, several granularities of tasks had been tested. This example shows that for tasks bigger than 500 ns the overhead due to the RAC is lower than 10% of the execution. As a comparative example, this task granularity corresponds to a mutex access in an embedded OS like µCOS II running in a ARM 926 at 200 MHz.

6. Conclusions and future works

In this paper we have discussed an auto-adaptive asynchronous parallel controller component implementing in 0.13µm CMOS technology. This module executes Petri nets that model the control flow of an application. This programming model is simple and enables to describe complex applications merging several kinds of parallelism.

The RAC is a trade off between the direct implementation of Petri net and the programming implementation found in the art. The features and the performances reached by the RAC allow new tradeoff between the dynamicity and the flexibility of the control flow and the performance of the system. The regular structure of the RAC provides it good scalability properties which have been verified in this paper.

To ease application development on RAMPASS, software tools able to take an application specified in a procedural language and able to split it in tasks handled by the RAC has to be studied. A board had been developed to validate the prototype. The circuit is coupled to a Virtex 4 on which can be implemented the operators managed by the RAC. The validation of the prototype is actually running.

7. REFERENCES

[1] Nomadik. Open multimedia platform for next generation mobile devices. Technicla report, ST Microelectronics, 2004

[2] Nexperia, PNX15xx Series Data Book – Connected Media Processor. Technical report, Philips, 2004

[3] João M. P. Cardoso, Self Loop Pipelining and Reconfigurable Dataflow Arrays, in International Workshop on Systems, Architectures, MOdeling, and Simulations, LNCS 3133, july 2004, pp. 234-243

[4] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths, Twentieth Anniversary Conference on Advanced Research in VLSI 1999

[5] R. David, D. Chillet, S. Pillement, and O. Sentieys. DART: a dynamically reconfigurable architecture dealing with next-generation telecommunication constraints, Int. Reconfigurable Architecture Workshop, Fort Lauderdale, April 2002

[6] PACT XPP - A self-Reconfigurable Data Processing Architecture, International Conference on Engineering of Reconfigurable Systems and Algorithm, Las Vegas, NV, June 2001

[7] A. J. Bernstein. Program analysis for parallel processing. IEEE Transaction on Electronic Computers, 1966

[8] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of Petri Net methods for controlled discrete event systems. Discrete Event Dynamic Systems, 1997.

[9] J. L. Peterson. Petri Nets. ACM Computing surveys, 1977

[10] T. Murata. Petri Nets: Properties, analysis and applications. Proc. of the IEEE, 1989

[11] S. S. Patil. An asynchronous logic array. MIT, Project MAC, Comp. Struct. Group Memo 111-1, 1975

[12] R. W. Hartenstein, A. Hirschbiel, M. Weber. Patil Array - a Petri Net Hardware Implementation, IEEE Comput. Soc. Press, CompEuro 88 - System Design: Concepts, Methods and Tools, 1988

[13] J. P. David. Architecture synchronisée par les données pour système reconfigurable, PhD thesis, Université catholique de Louvain, 2002

[14] S. Bulach. The design and realization of a custom Petri Net based programmable discrete event controller. PhD thesis, University of Ulm, 2002.

[15] Nicolas Ventroux, Stéphane Chevobbe, Frédéric Blanc, Thierry Collette. An Auto-adaptative Reconfigurable Architecture for the Control, Asia-Pacific Computer Systems Architecture Conference, Beijing 2004.

[16] Eva Dokladalova, Stéphane Chevobbe, Frédéric Blanc. Advances in Practical Implementation of the Digital Media Processing: Towards Reconfigurable Computation. European Workshop on the Integration of Knowledge, Semantic and Digital Media Technologies, London 2004