RAMPASS: Reconfigurable And Advanced Multi-Processing Architecture for future Silicon System

S. Chevobbe, N. Ventroux, F. Blanc, and T. Collette

CEA-List DRT/DTSI/SLA/LCEI
F-91191 Gif/Yvette FRANCE
phone: (33) 1-69-08-66-37
fax: (33) 1-69-08-83-95
Contact: {stephane.chevobbe,nicolas.ventroux,frederic.fblanc,thierry.collette}@cea.fr

***Abstract*— Due to the improvement of deep sub-microelectronic technologies, more and more transistors are available inside a die. But these improvements are difficult to exploit because of design complexity and time-to-market constraints. Thus, ASIC performances can not follow the microelectronic evolution. Regular, modular and reconfigurable architectures can easily take into account these evolutions because they are based on the repetition of small units.**

Reconfigurable devices are used to map different kinds of applications that exploit virtual hardware concepts. A more efficient utilization of such circuits is to adapt application at the run-time. During the process of an application, computing methods can change. Therefore, the next generation of reconfigurable architectures should provide many computing methods such as MIMD, SIMD, VLIW or multi-threading.

Any application is composed of two parts: a control part for operation scheduling and a computation part required for operators. Existing reconfigurable architectures use the same structure to implement these two parts. The solution presented in this paper is based on two reconfigurable resources. The first is suitable for control processes and the second for computation purposes.

***Index Terms*—dynamic reconfiguration,online reconfiguration, parallelism, reconfigurable architecture**

I. Introduction

Reconfigurable devices are composed of functional and interconnect resources. These resources can be arranged to implement a specific application. Semiconductor roadmaps indicate that integration density of regular structures (like memories) increase more quickly than irregular ones (cf. Table 1). So, reconfigurable architectures are suitable for future technology evolutions.

Table 1
Integration density for futur VLSI devices [1]

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

The key feature of reconfigurable architectures is the ability to perform hardware computations to increase performances, while retaining much of the flexibility of a software solution [2].

These architectures are very heterogeneous because their characteristics are adapted for many domains of applications (Co-processing acceleration [3], Hardware emulation [4], Fault tolerance [5], etc.). A classification based on the available resources of reconfigurable architectures, is presented below:

- The most famous fine grain reconfigurable architectures are **FPGA** (**F**ield **P**rogrammable **G**rid **A**rray). These devices merge two kinds of resources: the first one is an interconnection network and the second one is composed of processing blocks, based on special memories called **LUT** (**L**ook **U**p **T**able). Reconfiguration process consists in using the interconnection network to connect processing elements. Furthermore, each LUT is configured to perform the required operation.
- Some coarse grain architecture [5,6] has a reconfigurable network of interconnectors and an array of static processing elements. Processing methods depends on the network topology. For instance the RAPID architecture [6] is adapted for data-flow processing even though SYNTOL [5] is designed to perform **SIMD** (**S**ingle-**I**nstruction **M**ultiple **D**ata) processes.
- Some others architectures use a static network with

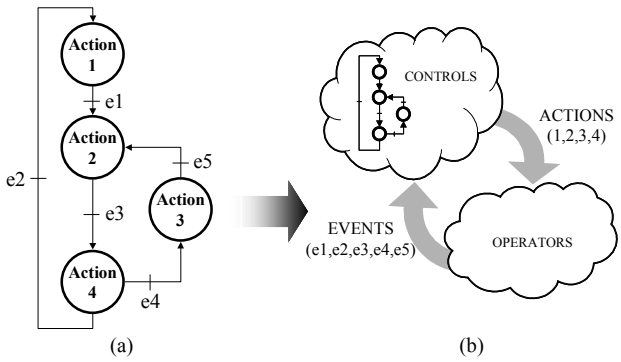


Fig. 1. Partitioning of an application in Control / Computation (a) Example of application – (b) partitionning

reconfigurable processing blocks. These architectures [7,8] have been developed to merge together a processor with reconfigurable units. Reconfigurable is used like extra ALUs. These architectures require powerful tools, which are able to manage **HW/SW** (**H**ard**W**are/**S**oft**W**are) partitioning [9].

- Besides, the DISC [10] architecture focuses on the duality between reconfigurations and instructions. Indeed, it is very hard to define the frontier between reconfigurable architectures and static ones. For instance, an instruction of a processor can be considered as a reconfiguration because it changes the data path for performing the required operation.

The main goal of all reconfigurable architectures consists in mapping a large set of applications. As shown in Fig.1, any application can be composed of two different parts:

- a control part for operations scheduling;
- a computation part for computing support.

These two parts have quite different characteristics for implementations. Indeed, the first one handles small data but requires global communications. On the contrary, the second one processes large data and uses local communications.

The reconfiguration frequency of these two parts are also different. In a processor, the control part is managed at each cycle whereas operators are fixed. Hybrid architectures like ONECHIP[7] or CHIMAERA[8] are based on a processor with extra reconfigurable operators. The instruction set is then adapted for a specific application. The reconfiguration frequency is quite low because the control is managed by the processor.

All existing architectures use the same structure to implement these two parts. Fine grain architectures are better adapted for performing control tasks than arithmetic tasks, which are well adapted to coarse grain architectures. In order to solve these drawbacks, some commercial architectures [11, 12] provide mechanisms that make arithmetic implementation more efficient. For instance, some multipliers are implemented in the VIRTEX [11] architecture. It might be advantageous to split reconfigurable resources into two parts. The first would deal with control processes, the second with computation purposes.

Within a given application, it is common to find different level of algorithms. For example, low-level image processing like convolution or filtering will often run on SIMD structures, whereas high-level ones as classification or face recognition are suitable for **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) architectures. Future architectures must provide many computing modes such as MIMD, SIMD, **VLIW** (**V**ery **L**ong **I**nstruction **W**ord) and multi-threading in a complex application. Although the physical hardware is limited, the virtual one can be much larger thanks to the reconfiguration process. The MATRIX [14] architecture focuses on this concept. In the respect, this paper will introduce new paradigms of architecture able to manage such kinds of parallelism processes (called **RAMPASS: R**econfigurable **A**nd **A**dvanced **M**ulti-**P**rocessing **A**rchitecture for future **S**ilicon **S**ystem).

II. FUNCTIONAL DESCRIPTION OF RAMPASS

In this section, the global functionality of RAMPASS is described. It is composed of two main reconfigurable parts (cf. Fig. 2):

- one dedicated to the control of applications called **R**econfigurable **A**rray of **C**ontrol (**RAC**);
- one dedicated to the computation named **R**econfigurable **A**rray of **O**perators (**RAO**).

In the first part, these two main elements are presented. In the second part, the working of the architecture is more detailed. Finally, the end of this section underlines the strong points of RAMPASS.

A. Overview

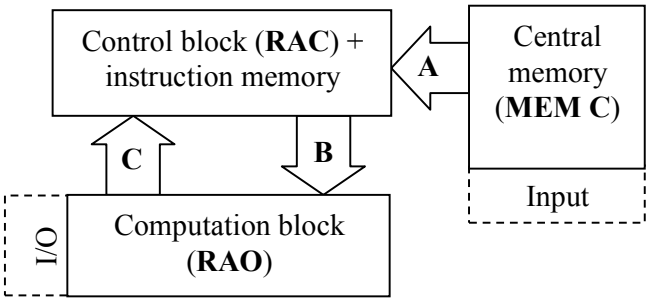
A **S**tate **G**raph (**SG**) is a classical model, which describes the control part of an application. This powerful model permits to represent complex computation concepts such as SIMD, MIMD, VLIW and multi-threading.

Thus, the first block can describe and store an application as a SG. These SGs are composed of states and transitions. States drive the computation elements in the RAO, and events coming from the RAO validate transitions in the SG. In order to allow SG implementations, this structure needs boolean logic and a powerful connection network.

An application described with SGs can be directly mapped in this block. Therefore, no translation in a specific language is required. The application remains user-friendly from the specification to the implementation.

RAC block can be seen as a cache memory instruction for reconfigurable elements. As in classical one, the efficiency of this block is ensured by code redundancies.

Whole SGs can obviously not be mapped in the RAC, so a mechanism as a dynamic reconfiguration has been introduced in order to permit to increase the virtual size of the architecture. Moreover, auto-routing mechanisms have been



Channels of communication :
A = Description of SGs and instructions
B = instructions and configurations
C = events

Fig. 2. Synoptic of RAMPASS

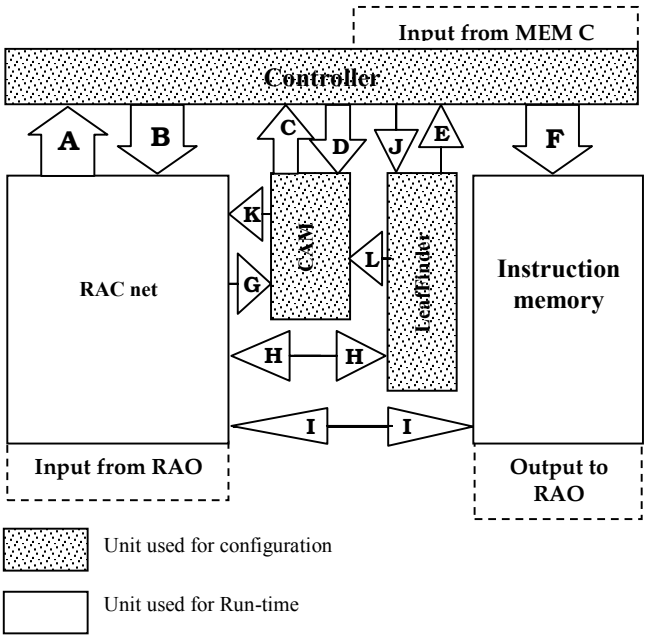


Fig. 3. RAC block

introduced in the RAC block to simplify SG configuration. Because the parallelism flexibility is ensured by the RAC part, we will focus on it and not discussed about the second block where computations elements are implemented.

B. *Mapping and running an application with RAMPASS*

In this part, the configuration and the execution of an application in RAMPASS are described. The application is stored in an external memory. A boot address must be defined to permit the boot of RAMPASS. Then, the SG is loaded in the RAC from this address. As soon as the SG begins to be stored in the RAC, its execution starts. Indeed, the configuration and the execution are simultaneously performed. During the execution, the SG stored in the RAC is continuously updated. The reconfiguration of the RAC is self-managed and depends on the application progress. This concept is called auto-reconfiguration and allows the architecture to self-manage its configurations.

The execution is based on few steps, which are initiated by the activation of a state stored in the RAC. When a state is activated (when a token is received), the associated instruction is sent to the RAO (cf Fig. 4). According to the instruction, the RAO returns an event to the RAC. This event corresponds to a transition in the SG mapped in the RAC. These transitions permit the propagation of tokens in SGs.

The architecture is globally asynchronous. Each block has its own mechanism of synchronization. Blocks are synchronized by acknowledgement protocols. The main protocol is between the RAC and the RAO because it controls the course of the application. RAC sends instructions and configurations to the RAO, which generates events.

C. *Strong points of RAMPASS*

Lastly, we focus on the main advantages of this new architecture. According to the application, computation grain is easily scalable. For example, a filter can be realized in at least two different ways. In the first place, the filter can be completely loaded and controlled by only one state, if the physical resources allow it. In the second place, simple operators as multiplier and adder can be physically stored in the computation block and driven by state graphs, stored in the control block. Tradeoff between computation grain and control has to be found for each application.

RAMPASS is especially dedicated to parallel algorithms. Because of the structure of the architecture, different kinds of parallel architectures can be loaded in RAMPASS as SIMD, MIMD, VLIW or multi-threading. Moreover, different kinds of reconfiguration allowed by RAMPASS decrease the resource limitation.

III. FUNCTIONAL DESCRIPTION OF THE CONTROL BLOCK: **RAC** BLOCK

In this third paragraph, more details of the control block are given. First, the different elements composing the RAC block are described. Then, the configuration and the execution of the RAC block are explained.

As previously introduced, the RAC is a reconfigurable block dedicated to the control of an application. It is composed of five units (cf Fig. 3): the *Controller*, the ***CAM*** (**C**ontent **A**ddressable **M**emory) and the *LeafFinder* are used to configure the *RAC net* and the *instruction memory*.

Although execution and configuration are running concurrently, it is important to understand the inner-functioning of each phase separately.

A. *Overview*

In this section, the five elements composing the RAC block are presented. These blocks are involved in auto-reconfiguration, auto-routing, and parallelism mechanisms.

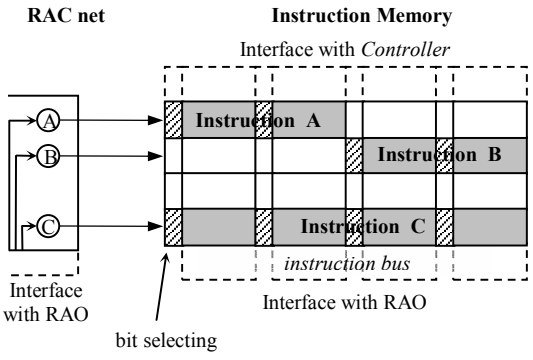


Fig. 4. Relation RAC net / Instruction memory

RAC net

This element is the core of the RAC block. It is composed of two resources: cells and interconnectors. The geometry of the SG is loaded in the *RAC net*. One state of SGs is implemented by one cell. The *RAC net* applies signals to the *instruction memory* to send the instruction stream to the RAO. The main characteristic of this unit is to physically implement the control of the application.

The *RAC net* is partially dynamically reconfigurable. Configuration and execution of SGs are fully independent. *RAC net* owns primitives to ensure the auto-routing. (*RAC net* will be developed in more details in the part IV).

Instruction memory

This memory contains instructions, which are loaded by the *Controller* for the RAO. To ensure parallel working, the *instruction memory* is scalable according to the length of the instruction. As shown in Fig. 4, the memory line is split in several words. Each line is separately driven by a state. The position in the line is given by the *Controller* during the storage (bit selecting).

Controller

This unit allows the connection between cells. It sends all the useful information to connect cells in the *RAC net,* which can auto-route itself. It can manage two kinds of connections between states:

- a new connection (the next state is not mapped in the *RAC net*);
- a connection between two states already mapped in the *RAC net*.

It also releases resources when the *RAC net* is full.

CAM

This element associates each cell of the *RAC net* used to map a state of a SG with its address in the external memory. It is used by the *Controller* 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 at the input of the *CAM*.

LeafFinder

This element identifies all the last cells of the active *SGs* mapped in the *RAC net*, which are called "leaf cells". It is a semi-mapped state which does not yet have an associated instruction.

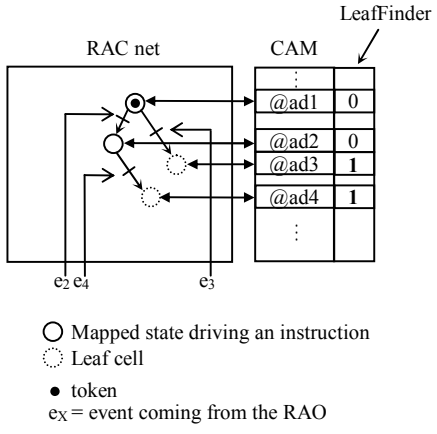


Fig. 5. Relation between *RAC net* / *CAM* & *LeafFinder* (note: relations with the *Controller* and *Instruction memory*)

B. Configuration

In this section, the operations of the different blocks involved in the implementation of a SG are presented.

All the application is stored in the external memory as a SG description. Each state description contains two kinds of information: the future instructions sent to the RAO and the descriptions of SGs. These future instructions will be loaded in the *instruction memory* by the *Controller*, and descriptions of SGs will be also loaded in the *RAC net* by the *Controller*.

Figure 5 shows the relations between *RAC net* and *CAM* & *LeafFinder* during the configuration of the *RAC net.* A simple OR divergence and convergence graph (cf . Fig. 6) have been chosen as an example. At this time of the course of the application, only a part of the SG is mapped in the *RAC net*: two fully states (state 1 and state 2), and two leaf cells (state 3 and state 4) are mapped. The *Controller* scans continuously the *LeafFinder* to detect a leaf cell in the *RAC net* (for example either state 3 or state 4). As soon as one is detected, the *LeafFinder* selects in the *CAM* its address in the external memory (for example either @ad3 or @ad4). This address is sent to the *Controller*, which reads the description of the corresponding state in the external memory. This description contains the configuration of the state and the addresses of its following states. In order to know whether the cell is already mapped in the *RAC net*, it supplies this address to the *CAM*. According to the result, the *Controller* sends the appropriate primitive to the *RAC net* to realize the connection (either a new connection, between state 4 and state 5, or a connection between two states already mapped, between state 3 and state 4). The *RAC net* notifies the *Controller* when the connection has succeeded. Then, the *Controller* ensures the update of:

- the *CAM* with the address of the new mapped state;
- the *LeafFinder* to define the new cell as a leaf cell;
- the *instruction memory* with the correct instruction.

When a connection fails, the *RAC net* indicates an error to the *Controller*. The *Controller* deallocates resources in the *RAC net* and searches the next leaf cell with the *LeafFinder*. These two operations are repeated until a connection succeeds.

C. Run-time

The two elements implied in the run-time are the *RAC net* and the *instruction memory*. As a cache memory, the *RAC net* "decodes" the address of the next instructions which will be sent to the RAO.

As shown on figure 4, a cell, which implements a state, is associated with an instruction stored in the *instruction memory*. When the active state (marked by a token) receives its associated event, its related instruction is sent to the RAO. The split instruction bus, presented upper, allows different kinds of parallelism introduced in the first part. For example, the instruction A and B could be sent together to different operators mapped in the RAO without creating conflict,

whereas the instruction C would be sent alone.

When the operation has finished, the cell transmits its token to the following cells.

IV. Description of RAC net

In this section, more details of the core of the RAC block are presented.

The *RAC net* works as an address decoder in implementing SGs. This network permits to map physically the control of an application using connections of cells. Its structure is a network based on the duplication of two basic elements: cell and interconnectors.

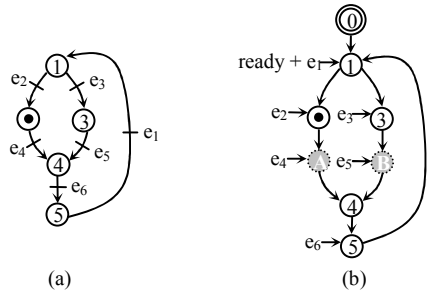
Elementary connections, which can be mapped in the *RAC net*, are presented in Tab. 2. Linear connections and divergences (AND and OR) are simple to realize. These kinds of connections do not require additional cell to be implemented. Divergent and linear connections are initiated by one cell to find paths to new cells, whereas convergences are initiated by several cells in order to connect one cell. So, the convergent cell must be at the intersection of the different initiating paths. That is why convergences are complex to map in the *RAC net*.

As presented upper, the *RAC net* is auto-routed and can map the different kinds of simple graph elements introduced in the Table 2, since six modes have been defined inside cells.

Two kinds of networks have been also introduced:

- one dedicated to the token propagation which connects cells together;
- one dedicated to events propagation coming from the RAO.

So, the network for tokens must allow boolean operations in order to implement OR and AND divergences. Reliability and flexibility of the *RAC net* depend on the number of allowed connections. Indeed, the higher the number of connections are allowed, the better the architecture can adapt itself to a large variety of SGs. This network is able to find and create paths between states already mapped (cf Fig. 5 between states @ad3 and @ad4), or create paths between leaf cells and free cells to implement new connections between states (cf Fig 5 between state @ad4 and a future state – not represented). This is an important characteristic of our network. The user has just to describe the geometry of SG without taking care of the routing in the *RAC net*. This routing is automatically done by the network itself. The *Controller* only gives the description of the geometry of SGs, since protocols between cells make possible the auto-routing. Upon a *Controller* request, a cell is able to find a free connectable neighbour and establish a connection. Only three signals are required to perform these operations.



Note: The marked state is state 2

supplier cell of token

supplementary cell needed for the implementation

Fig. 6. (a) State graph – (b) implantation of the state graph in the RAC net

Besides, the network for events is a little less constraint. But it must be sufficiently flexible to allow connections from events coming from the computation block to each cell of the *RAC net*.

TABLE 2
ELEMENTARY IMPLEMENTABLE PART OF SGS

Possible geometry	Cost in RAC cell vs. real graph	Interconnection complexity
	add none	very simple
	add none	simple
	add n cells for n convergences (OR)	complex
	add none	simple
	add n cells more for n convergences (AND)	very complex

V. Simulations and results

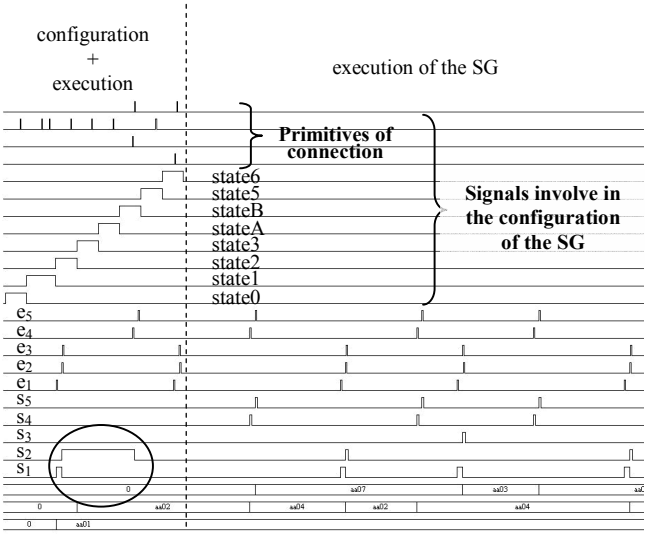
A functional model of the RAC block has been realized with SystemC. In this part, the results of this model are given, and examples of implementations in the RAC block are discussed.

The characteristics of this description language easily allow hierarchical projects. Although SystemC is a hardware description language, it has the flexibility of the C++ language [16] and owns its paradigms.

A lot of different programming structures have been implemented in the RAC block, e.g. exclusion mechanisms, synchronisations between separated graphs, etc. Moreover, an application of image processing (complex motion detection) has been mapped. These results permit to consider interesting performance improvements for future works in image embedded systems. Furthermore, these simulations have permitted to validate the paradigms of RAMPASS as auto-routability and auto-reconfiguration during execution of SGs.

In this paper, only a simple graph presented in Fig. 6 is the subject of a particular study. The chronogram (cf Fig. 7) represents its simulation. It shows mapping and execution steps of a simple state graph. The first part of the chronogram clearly shows the overlap of the configuration and the execution of the SG. Indeed, stateX signals and s_1, s_2 signals move simultaneously. StateX signals, which indicate the end of

the implementation of a state, are involved in the configuration. s_X signals correspond to the presence of the token in a cell. The second part of the chronogram shows the progression of the token in the SG. This propagation through the two branches of the SG is driven by events (e_x). The last three lines illustrate the concept of the split bus. Indeed, each



e_X = event from the RAO (involves in the execution of the SG)
s_X = signal illustrate the presence of a token in a cell (involves in the execution of the SG)
stateX reports that a state is implemented in the *RAC net*

Fig. 7. Chronogram during the implementation and the execution of a simple SG (cf. Fig. 6 (b))

part is independently managed and shared between all states.

VI. Conclusion and Future Works

New architectural concepts are proposed in this paper. A reconfigurable part dedicated to the control of applications is detailed. This control is described using a state graph representation. To perform this control, RAMPASS uses its RAC net as an address decoder. This innovation consists in using a reconfigurable technology to solve drawbacks of cache memory. Thanks to state graph characteristics and the concept of auto-routing, this structure is able to perform many kinds of parallelism processes as SIMD, MIMD, VLIW and multi-threading. The concepts of the RAC block have been successfully checked by simulations.

According to the encouraging results, further works will be performed. To evaluate performances of RAMPASS, a Cmos model and a prototype of the RAC block will be made. But before, the topology of the interconnection network (RAC net) will be optimised. At the same time, a modelisation of the RAO block will be performed.

References

[1] H. Nakada, K. Oguri, N. Imlig, M. Inamori, R. Konishi, H. Ito, K. Nagami et T. Shiozawa, "Plastic Cell Architecture : A Dynamically Reconfigurable Hardware-based Computer", *in Proc. of IPPS/SPDP'99, Springer Verlag, LNCS 1586*, pp.679- 687, April 1999.

[2] K. Comptin, S. Hauck, "Reconfigurable Computing : A Survey of Systems and Software" in *ACM Computing Surveys*, Vol. 34, No 2, June 2002, pp. 171-210

[3] J.R.Hauser, J.Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor" in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '97), April 1997

[4] K.S.Oh, S.Y.Yoon, S.I.Chae "Emulator Environement based on an FPGA Prototyping Board" in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping* (RSP 2000)

[5] F.Clermidy, T.Collette, M.Nicolaidis, "A new placement algorithm dedicated to parallel computer", *in Pacific Rim International Symposium on Dependable Computing*, Dec 99

[6] C.Ebeling, D.C.Cronquist, P.Franklin "RaPiD - Reconfigurable Pipelined Datapath" in *The 6th International Workshop on Field-Programmable Logic and Applications,* 1996

[7] R.D.Wittig and P.Chow "OneChip: An FPGA processor with reconfigurable logic" in *Procedeedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (FCCM'96), pages 126-135, 1996

[8] S.Hauck, T.W.Fry, M.M.Hosler, J.P.Kao "The Chimarea reconfigurable functional unit" in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (FCCM'97), pages 87-96, 1997

[9] Y.Li, T.Cattahan, E.Darnel, R.Harr, U.Kurkure, J.Stockwood, «Hardware-sofware codesign of embedded reconfigurable architectures» *in 37th ACM/IEEE Design Automation Conf., L.A.,USA*, June 2000.

[10] M.J.Wirthlin, B.L.Hutchings "DISC: the Dynamic Instruction Set Computer" in *Proceedings of the SPIE Reconfigurable Computing Conference: Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, pages 92--103*, Oct 1995

[11] Virtex II www.xilinx.com

[12] Stratix familie www.altera.com

[13] S.Copen Goldstein, H.Schmit, M.Budiu, S.Cadambi, M.Moe, and R.Taylor, "PipeRench: A Reconfigurable Architecture and Compiler" in *IEEE Computer, Vol.33, No. 4,* April 2000

[14] E.Mirsky, A.Dehon "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996

[15] "The Open SystemC Initiative" www.systemc.org/overview.htm

[16] S. Y. Liao, "Towards a new standard for system-level design", in *proceedings of the Eighth International Workshop* on Hardware/Software Codesign, 2000.