Dynamic parallel reconfiguration for self-adaptive hardware architectures

Laurent Fiack[1], Benoît Miramond[1], Andres Upegui[2], Fabien Vannel[2]
[1] ETIS Lab UMR 8051 CNRS / ENSEA / UCP
6 Avenue du Ponceau, 95014 Cergy-Pontoise, FRANCE
Phone: +33 1 30 73 66 10
Email: {firstName.lastName}@ensea.fr
[2] Institute of Informatics and Telecomunications - inIT,
University of Applied Sciences Western Switzerland, hepia, HES-SO
Rue de la Prairie 4, 1202 Geneva, Switzerland
Email: {firstName.lastName}@hesge.ch

***Abstract*—Adaptive Hardware Systems can rely on software or hardware adaptation. Software adaptation can be globally assimilated to mode switching, either at a technological or hardware level (DVFS, Idle processor mode ...), or at the application level (bandwidth adaptation in telecommunication, multispectral cameras, ...). Hardware adaptation corresponds to a deeper change in the internal organization of the computing architecture of an embedded system. It enables more powerful adaptation but is currently limited by the reconfiguration (tool and architecture) of today's FPGA devices. We present in this paper a multi-FPGA platform designed to exhibit unique computing capabilities. The joint design of the electronic board and the internal architecture of each reconfigurable device permits dynamic parallel (and not partial) reconfiguration of several parts of the system while maintaining global routing and local computation in the rest of the system. Dynamic parallel reconfiguration and technological independence are enabled by considering reconfiguration at coarse grain. We describe in the paper the hardware elements composing the platform. The specific design of the global system allowed us to reach a fully operational platform. We present statistical experiments to evaluate the inter-chip network capacity which show that our platform supports up to 18 parallel reconfigurations per second.**

I. Introduction

During recent years the integrated circuit industry has experienced a tremendous growth in terms of number of transistors but also, and less expected, in terms of novel applications and computation paradigms. This complexity enhancement due to the increased level of integration has enabled the appearance of multiprocessing systems as a commercial solution for consumer end electronics. This trend will certainly continue and in a near future we will be dealing with integrated circuits containing a large array of processors.

This new overall system architecture imposes a completely new programming paradigm. Given the increase in the level of system complexity, the programmer cannot keep a full control of the program execution, the hardware must be able to self-adapt in order to dynamically decompose and allocate tasks and sub-tasks execution, and efficiently route information between these tasks. Moreover, according to computation needs, the number of tasks may evolve at run-time. According to environmental or external factors such as incoming data-load or the richness of incoming sensor data, some tasks may require more computing power than others at a certain moment. Computing resources must thus be relocatable in order to satisfy computation needs, in terms of real-time constraints or, in general, the desired achievable performance.

978-1-4799-5356-1/14/$31.00

In this paper we present a scalable self-adaptive multi-processor system architecture, which supports the above-described computation needs. The overall architecture is composed of an array of processors each with it's own instructions and data memory and peripherals. A MSP430 soft-core processor is used as a general purpose computing unit becoming a computation cell making part of a cellular array, it may be replaced by any other processor. This computation cell lies on a distributed routing cellular array. Computing resource allocation is critical in this kind of platforms, different tasks with diverse computing requirements may require a particular hardware architecture in order to be implemented in an optimal manner. The reconfigurability of the distributed system presented in this paper allows to do so and proposes a strategy for efficiently dealing with dynamic task allocation.

The main interest of the presented architecture lies on the heterogeneity of the computing modules supported by a distributed reconfigurable platform. This aspect enables hardware to dynamically adapt to applications with computation requirements. Dynamic hardware reconfiguration enables dynamic hardware-based task allocation. A good example, in which we are in fact currently working on, is video processing [**?**]. Visual information analysis is dynamic by nature: according to the complexity exhibited by a static image, one may need more or less computing resources in order to perform an image-content analysis, motion information will also demand computing resources. In the human brain, visual dynamic stimuli is analyzed very differently than static stimuli. Stimuli perception is localized on clearly identified areas of the brain cortex, which exhibits important activity according to the nature of the observed pattern [**?**]. This task separation on the

visual cortex has inspired us for proposing a reconfigurable architecture supporting two different types of processors: a static-vision analyzer and a motion analyzer. According to the nature of input visual stimuli, the available computing resources will be configured with a certain number of static-vision analyzers and motion analyzers. Both processors will thus compete for the available reconfigurable resources in order to deploy as much analyzers, according to the richness of the analyzed features present in the input data.

Previous works on system prototyping based on multi-FPGA systems had mainly focused on design partition of very large systems that cannot fit into a single device [**?**]. This use of multi-FPGAs doesn't exploit their reconfigurability nature since the system architecture remains static after configuration. Previous efforts towards dynamic hardware allocation on multi-FPGA environments have been done in [**?**], in his dissertation the author describes a cellular system based on a set of homogeneous computing nodes in the form of MOVE processors, communicating through a routing layer based on HERMES [**?**]. The system has been implemented on the Confetti platform [**?**], which is also used by the work presented in this paper and is described in detail in the next section.

This paper presents thus a new approach for self-adaptive Network on Chip (NoC) multi-processing system deployment. Heterogeneous computing nodes can be reconfigured in a distributed and autonomous manner according to the computation needs of the task at hand. We present the overall system architecture, its deployment on a multi-FPGA platform, and we present some results about the inter-node communication capabilities between NoC's components. Finally, we conclude and discuss our vision of future research on this direction.

II. Confetti: a multi-FPGA platform

Confetti is a multi-FPGA platform specifically designed for prototyping cellular architectures. The platform is composed of a set of stacks of printed circuit boards (PCBs), that can be connected together side by side to create two dimensional arrays of arbitrary size (Fig. **??**).

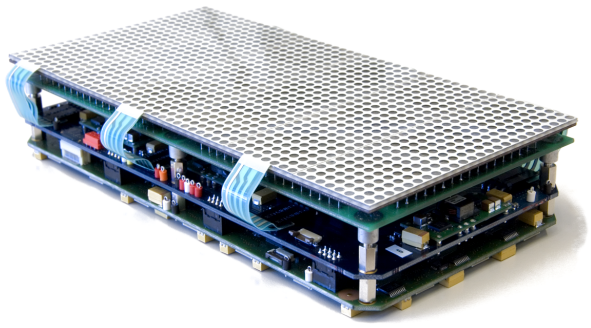

Fig. 1: **Confetti stack:** A single module composed of 18 Cells. Several stacks can be linked to expand the cellular surface.

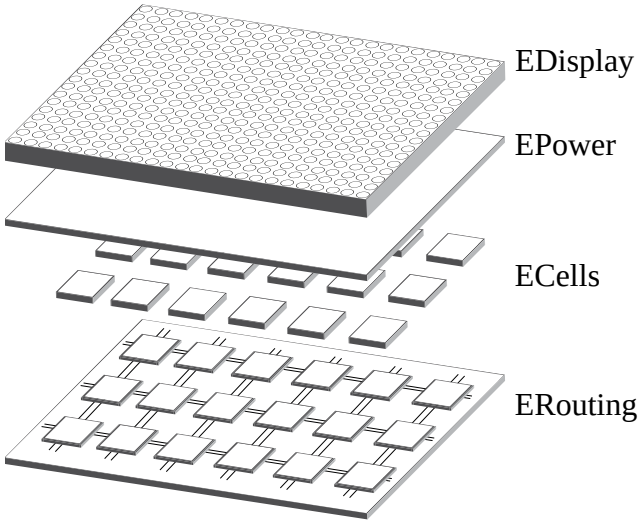

Fig. 2: **Confetti stack:** Architecture of the 4-layers Confetti stack.

A. An architecture devoted to cellular computing

Confetti's logic architecture is composed of the four layers presented in Fig. **??**: a computation layer, a routing layer, a sensitive/display layer and a power layer.

The computation layer is composed of an array of FPGAs - called *ECells* - and it is in charge of performing the computation of the cellular system. Each *ECell* can be configured with different FPGA configurations and can also be reconfigured in runtime. The platform can support heterogeneous multi-processor architectures.

The routing layer is composed of a second array of 18 FPGAs (for one stack) - called *ERouting* - and it manages the connectivity among the computation cells. The board implements a routing network based on a mesh topology which provides inter-FPGA communication but also communication with other routing boards.

The *EPower* board manages the power supply of all the stack and the startup sequence.

The top layer, *EDisplay*, is the user interface. It is composed of an RGB LED matrix and an array of touch sensors providing interaction capabilities.

Each module can be considered as a cellular architecture made of several layers, as depicted in Fig. **??**. All the routing FPGAs are configured with the same configuration and are in charge of two major tasks: inter-communication between each *ECell* FPGA and dynamic configuration of *ECells*.

Each routing FPGA is directly connected to a 16 Mbit FLASH memory, which is used to store *ECell*'s configuration and non-volatile data for computational applications. Since each *ECell* FPGA bitstream size is 1 Mbit, these memories allow up to 16 *ECell*'s configurations to be stored by each Confetti cell.

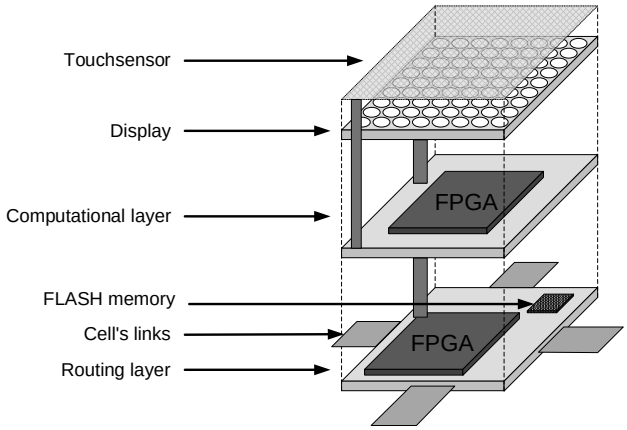

Fig. 3: **Confetti cell:** The *ECell* is the applicative FPGA and is directly linked to the *ERouting* FPGA which can communicate in four directions.

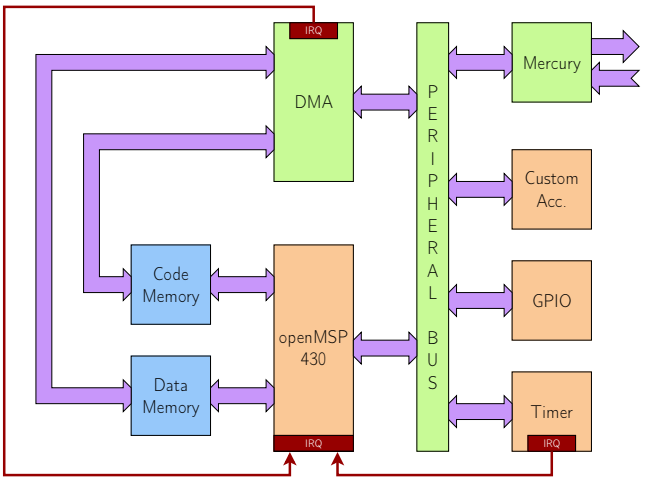

Fig. 4: **Architecture of a computation cell.** The computation cell is composed of an openMSP430 processor, local memories, a network interface and a DMA.

Since every *ECell* FPGA can have a different configuration and even be dynamically reconfigured during system operation, one of the main problems to be addressed is how to redirect the desired configuration to each of the *ECell* FPGAs in the system.

This task, which is performed within the *ERouting* board, will be presented in the following sections.

B. The dynamic configuration as a routing challenge

The first version of the platform didn't allowed to share *ECell*'s configuration among the Confetti platform. All the FLASH memories had to have the same content. In the work presented in this paper, we add self-configuration capabilities to the global system. An *ECell* can interact with the *ERouting* layer to trigger several types of dynamic parallel configuration:

- self-reconfiguration with one of the 16 configurations stored into the local FLASH.
- self-reconfiguration with a configuration stored on a different *ERouting* FLASH.
- reconfiguration of another *ECell* or a set of other *ECells*. The configuration stream should then be streamed through the communication interface of the *ERouting* layer.

Physically, every *ERouting* FPGA is linked to its four cardinal neighbors and to the ECell board above it (Fig. **??** and **??**). This setup was selected for modularity and scalability purposes. The links between each FPGA are implemented using the built-in LVDS (Low Voltage Differential Signaling) drivers that allow, in our case, data rates up to 500 Mbits/s. Two communication buses (one for each direction, 3 bits per bus) are present for each neighboring pair. Since there is no global clock in the system and no clock recovery possibility, a clock signal is transmitted on one differential pair to synchronize the data transmitted on the two other pairs. Thus, at *ERouting* level, a bandwidth of 1 Gbit/s is available on each *ERouting* FPGA for every direction. Moreover, the same type of bus exists between each ERouting FPGA and its corresponding *ECell*. As the *ERouting* boards constitutes the communicating backplane of the whole Confetti, connections between the different stack boards are also implemented here. External connectors are present on the four sides of the board and provide the same connectivity as the links between the FPGAs: two adjacent ERouting boards then represent effectively a single uniform surface of FPGAs. This setup allows the creation of systems consisting of several stacks that behave as a single, larger stack.

In order to store new configuration into a FLASH, an external interface connects as an adjacent stack and allows to interact with a PC using USB link. It is then possible to stream a new configuration to all FLASH memories or to a specific one. Some bytes are added to the configuration stream to add informations, like configuration name, storage address, ... Currently it takes a few seconds to store a new configuration into a FLASH of the Confetti tissue, but it only takes 20 ms to configure a *ECell* FPGA from one FLASH, using self-configuration method. The *ERouting* layer can stream FPGAs configuration coming from another FLASH cell. Time of the operation depends on the network usage and will be discussed in next sections.

III. Internal architecture of the computation cells

In this section, we describe the internal architectures contained by the ECell and the ERouting FPGAs introduced in Section **??**.

Each ECell hosts a microprocessor system coupled with a communication device. The ERouting FPGAs implement a router, communication devices and a few control hardware.

A. The computation tile

The computation tile, as depicted in Fig. **??**, is built around an openMSP430 16-bits processor from the OpenCores library[**?**]. The choice of this synthesizable microprocessor is

motivated by two main aspects: technology independence and open access to the source code.

This allows us to foresee experiments of the self-reconfiguration mechanism on other platforms. Moreover we want to be able to modify all soft IPs to fit our needs. For these reasons we decided to adopt an open source microprocessor.

Secondly, the openMSP430 is a mature project and the code is well documented, so we can easily embed it into our tile and adapt it by adding custom peripherals. Finally, as its instruction set architecture is compatible with Texas Instruments' MSP430 microcontroller family, it can execute the code generated by an MSP430 toolchain from high-level languages, which makes the software development easy.

The openMSP430's major drawback resides in the fact that the different buses don't allow to be interrupted by another master. This point makes the development of DMA uneasy.

However, as communications are of major importance for our project, the use of a DMA block is mandatory. To get over this impediment, we benefit from the dual-port RAM of modern FPGAs and designed a dedicated DMA-capable bus for the concerned peripherals.

The DMA is connected to both instruction and data memories to allow, on one hand data transmission, and on the other hand, the instruction sequence modification of a processor by one of its neighbors. As the DMA can read the code memory, a tile is capable of replicating its code into another tile.

We designed the DMA to manage the memory as a FIFO. The hardware and the software parts share the read and the write pointers. During data reception, the hardware increments the write pointer and is aware of the software read pointer to avoid writing while the FIFO is full. When the software part reads the FIFO, it increments the read pointer and verify the emptiness of the FIFO before reading.

When the FIFO is full, the DMA indicates the network interface to halt the reception. In the other direction, the DMA can obviously be halted by the network interface.

The transmitter we use was developed by Mudry in [**?**] during the initial design of the Confetti platform. The transmitter is composed of a 32-bit shift register to serialize the data before the inter-FPGA asynchronous communication (see figure **??**). The receiver is composed of a 32-bit shift register to de-serialize the data coupled with a dual-clock FIFO to synchronize the clock domains.

The interface, depicted in Fig. **??**, consists in three wires: *clock*, *data* and *ready* signals from the receiver. These signals are sent through an LVDS buffer.

B. The router

The *Mercury* transceiver allows communications between FPGAs with a reduced number of wires, but only to a local neighborhood. The ERouting FPGAs are intended to allow global communications by hosting dedicated hardware to route data from every ECell to every other.

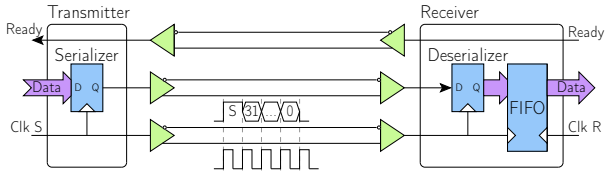

Fig. 5: **The hardware of Mercury sender and receiver.** *clock* and *data* signals are used to send data from the transmitter to the receiver. A *ready* signal indicates the transmitter that the receiver is ready to accept data.

Thus, the ERouting board can be seen as a Network-on-Chip (NoC) infrastructure IP. This NoC follows a mesh topology which makes it easily scalable.

We use the same NoC as Mudry in [**?**] which is based on the *Hermes* framework [**?**]. The *Hermes* NoC uses packet-switching where the packets include a header with the destination address and the packet length.

The *Hermes* NoC is composed of switches with five data ports. One is the local port, which is connected to the associated ECell. The others are the external ports, which are connected to the neighbor switches in the four cardinal directions.

Switches allow point-to-point communication between any pair of ECell in Confetti by using the address included in the header to redirect the packets.

The *Hermes* NoC uses wormhole switching where the packets are divided into flow control digits (flits). Each *Hermes* switch stores only one flit, called the header flit, which contains the routing information, such as the destination address and the packet length.

The decoding of the header flit allows the creation of a dynamic path where the following flits are shifted in a pipelined way.

There are a few differences between the original *Hermes* framework and Mudry's implementation. First, the flits are serialized thanks to *Mercury* transceivers so that the amount of wires between FPGAs remains acceptable. Secondly, the Confetti platform is designed following a Globally Asynchronous Locally Synchronous (GALS) paradigm. This type of communication is needed to reach scalability of the platform and is enabled by the *Mercury* transceivers too.

In addition, we extended the design with a configuration logic block that enables the configuration from the ERouting FPGAs to the associated ECell FPGA. The configuration logic propagates the bitstream to neighbor FPGAs and to a local Flash memory. The ECell FPGA can be configured with the bitstream included in the Flash.

Finally a GPIO logic communicates with the associated ECell to give the address, and the state of the touch screen and get the state of the screen.

The ERouting design is depicted in Fig. **??**.

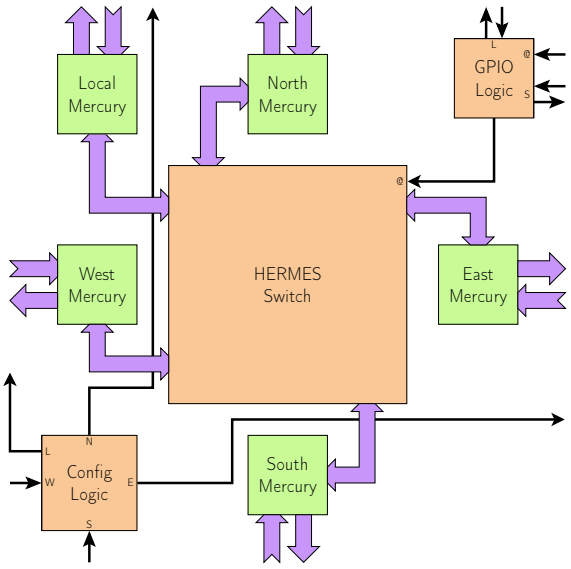

Fig. 6: **Architecture of a routing cell.** The switches allow point-to-point communication between any pair of ECell in Confetti by redirecting the packets.

IV. Experiments and results

This section presents a qualitative validation of the system through the implementation of a cellular automata. We then discuss some quantitative measures to characterize our adaptation of the Network on Chip (NoC) described in **??**.

The first validation was achieved by implementing a Conway's Game of Life where the cells are organized in a 2D grid [**?**]. Each cell is represented by an LED of the screen and can have two possible states, alive (LED On) or dead (LED Off).

Each computation cell manage a 8×8 grid and shares its edges with its neighbors that enable the emergence of a global automata. One can think that in this case the routing is unnecessary, but it allows our implementation of the automaton to be toroidal.

A mechanism of blocking `read` and `write` allows the synchronization of the automata (*ie.* all the tiles are in the same iteration). During a `write`, the DMA is set to send the buffer to the *mercury* interface, leading to the start of the communication. On the receiver's side, the DMA automatically stores data into a buffer. A call to `read` copies the buffer when it's not empty.

This simple experiment requires the main functionalities of Confetti (display, touch screen and communications). Moreover a test with two EStacks validated the scalability potential of the platform.

A. Description of the experiments

Confetti – as the BioWall successor [**?**] – is a unique platform. Therefore, it is not straightforward to find a point of comparison. Furthermore, there is no point in comparing our system's throughput and latency with those of NoCs or supercomputing networks as the aim of this platform is to prototype new paradigms rather than target high performance computing. The purpose of these experiments is to link the throughput to the openMSP430's programming time and the FPGAs reconfiguration time.

We ran two experiments to measure the network capacity and to predict reconfiguration times:

1) Communications in a pair of adjacent ECells,
2) Communications in an EStack (3×6 ECells).

The first experiment aims to find out maximal achievable throughput and the minimal latency. One ECell proceed to a sequence of *pings* toward a direct neighbor with different packet sizes. This allows to set apart the throughput from the latency.

These measures will allow us to extrapolate the configuration time of a full ECell FPGA from its neighbor, and the programming time of an openMSP430.

This experiment was performed three times with different receiver's strategies:

1) It copies the entire buffer and sends it back,
2) It discards data, but increments the read pointer,
3) It discards data and waits the entire message before incrementing the read pointer.

This brings out the impact of the receiver on the transmissions.

The second experiment aims to show up the capacity of the network. It is divided into time slots where each ECell decides whether or not to send a packet by picking a random number following the uniform law in $[0,1]$. If this number is below a threshold – called *injection rate* – the ECell's processor sends a packet. It then transmits until the end of the slot. If it is not allowed to send a packet, it waits until the next slot. The coordinates of the destination follows the uniform law too. We observed the throughput and the latency in function of the injection rate.

If at the end of the slot the ECell was not able to send its packet, the packet is considered lost.

The packet contains a time stamp so that the receiver can compute the latency by comparing with its own time. The throughput, as defined in [**?**], is computed as follows:

$$TP = \frac{(\text{Total messages completed}) \times (\text{Message length})}{(\text{Number of ECells}) \times (\text{Total time})}$$

where the *Message length* is measured in bytes, and the *Total time* in cycles. We think the time measure in cycles is more relevant as it doesn't depend on the technology evolution. Thus the throughput is measured in bytes/cycle/ECell. The measures in seconds will be given for the *Spartan III* as well.

B. Results

The three *ping* experiments show the following results at 20 MHz:

1) When the entire buffer is copied and sent back, we measured a throughput of 833 kB/s (0.04 B/Cycle) and an extra latency of 19 µs (378 Cycles),
2) Without the copy, the throughput is 1.6 MB/s (0.08 B/Cycle) and the latency is 16 µs (316 Cycles),
3) Finally, when the message is ignored by the receiver, the throughput reaches 2.2 MB/s (0.1 B/Cycle) and the latency drops to 7.7 µs (154 Cycles).

The knowledge of the throughput allows us to foresee the configuration time of a full ECell FPGA from a direct neighbor. The configuration of an ECell is performed by the corresponding ERouting FPGA through a dedicated hardware. As the impact of this hardware will have a negligible impact on the latency, we can take the throughput obtained in experiment 3).

A *Spartan 3* bitstream is 128 kB wide, leading to a bitstream communication time of

$$\frac{128\,\text{kB}}{2.2\,\text{MB/s}} = 60\,\text{ms}$$

This time is slightly higher than the 20 ms required for the reconfiguration of a *Spartan 3* in Confetti, but it is in the same order of magnitude.

The software programing time through the network can be deduced the same way. Currently the code size is set to 4 kB and can't exceed 64 kB due to the 16-bit bus of the openMSP430. This leads to a code communication time of

$$\frac{64\,\text{kB}}{2.2\,\text{MB/s}} = 30\,\text{ms}$$

The second experiments shows the limits of the network due to the congestions.

The throughput, as shown in Fig. **??**, grows linearly in function of the injection rate until around 0.34 %. The throughput ceils at 140 kB/s/ECell (0.007 Byte/Cycle/ECell at 20 MHz).

The latency, as depicted in Fig. **??**, grows exponentially until a threshold at an injection rate of about 0.4. This is due to the fact the messages are deleted when the network is busy during the entire send slot. If not, the latency would be boundless.

We can foresee again the bitstream communications:

$$\frac{140\,\text{kB/s/ECell}}{128\,\text{kB}} = 1.07\,\text{Reconf/s/ECell}$$

Thus our architecture enables up to 18 parallel configurations per second for one Confetti stack.

To conclude, the developer should keep in mind the limits of the network. To decrease the network congestions, it's advised to communicate with direct neighbors. The bitstreams and openMSP430's codes should particularly be sent locally.

A broad/multi-cast system should be added shortly to reduce congestion problems.

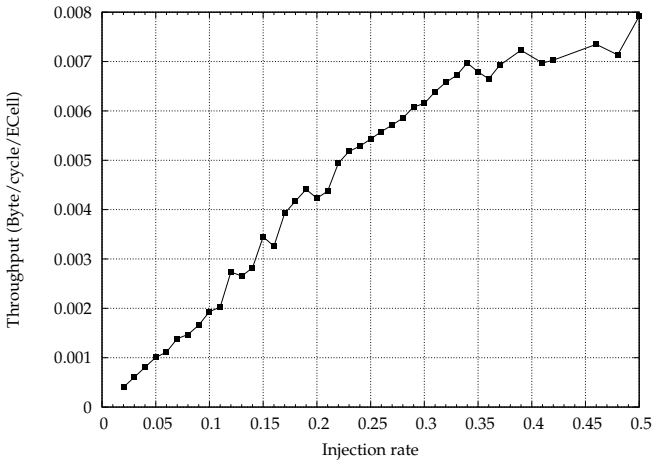

Fig. 7: **Throughput versus Injection rate**

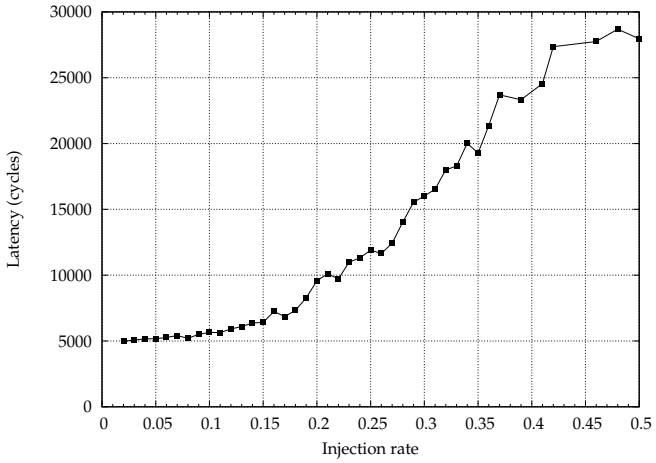

Fig. 8: **Latency versus Injection rate**

V. Future work

The platform presented in this paper has been developed in the context of the RETINE project [**?**]. The goal of the project is to enable self-adaptive hardware for autonomous mobile robotics. The array of ECells, composing our platform, is the main computing substrate of the robot and is thus shared between the tasks processing the afferent signals coming from the sensors of the robot: vision, auditory system, proprioception, tactile sensing, vestibular system... Hardware adaptation, through dynamic parallel reconfiguration, is used to adapt the internal organization to the richness of the incoming information. Thus, the embodied computer develops itself to process multiple sensory inputs in order to use effectively its body in the environment.

This adaptive mechanism is implemented as a learning process controlled by an unsupervised neural network distributed into each cell of the platform [**?**]. The neural network learns at the rhythm of the environment and so exhibits concurrent behaviors for each of its entries. At the hardware level, this concurrency (or competition) between sensing data relies on

parallel computation and reconfiguration. The first results of this bio-inspired adaptation mechanism have been presented in [**?**]. We are now working on the deployment of the neural network using the dynamic parallel reconfiguration offered by the proposed platform.

VI. Conclusion

We presented in this paper a scalable self-adaptive multiprocessor system designed to experience bio-inspired computing architectures. The properties of the system can be reached thanks to a specific design at two organization levels. Firstly, the platform is composed of several FPGA devices locally connected through asynchronous communication links. This connectivity and the clear separation between computation cells and routing cells bring the scalability required to deploy massively parallel reconfigurable architectures. Secondly, the internal architecture of each cell has been designed specifically to support programmability of *Ecells* devices and inter-devices routing capabilities of *ERouting* devices.

So, bio-inspired computing models can be tested and validated thanks to a high-level programming interface using a message passing protocol. Once a model deployed onto the platform, computing cells can be dynamically reconfigured concurrently while the system still ensures communication between running cells.

This property is unique compared to dynamic partial reconfiguration (DPR) of single FPGA devices, most often used to validate adaptive hardware systems. Moreover, the proposed platform exhibits these novel capabilities in reconfigurable and adaptive computing without technological dependencies on the DPR synthesis tools from FPGA vendors.

This paper focused on the hardware infrastructure needed to reach such an operational platform and presented experimental results on the routing challenge induced by dynamic parallel configuration. We are now working on the deployment of neuro-inspired computing paradigms [**?**] in the context of artificial vision for autonomous mobile robots.