A hardware-software design framework for distributed cellular computing

Pierre-André Mudry[1], Julien Ruffin, Michel Ganguin, and Gianluca Tempesti[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL),
Biologically Inspired Robotics Group (BIRG)
Station 14, CH - 1015 Lausanne, Switzerland
[2] University of York, Department of Electronics,
Heslington, York YO10 5DD, United Kingdom

Abstract. In this article, we describe a novel hardware-software design framework for prototyping cellular architectures in hardware. Based on an extensible platform of about 200 FPGAs, configured as a networked structure of processors, the hardware part of this computing framework is backed by an extensible library of software components that provides primitives for efficient inter-processor communication and distributed computation. This dual software–hardware approach allows a very quick exploration of different ways to solve computational problems using bio-inspired techniques. To demonstrate the validity of the method, we present an example of how a traditional parallel system such as a cellular automaton can be modeled and run with this perspective. In addition, we also show that the flexibility of our approach allows not only cellular automata but any computation to be easily implemented on a cellular substrate.

1 Introduction and motivations

The application of bio-inspired mechanisms such as evolution, growth or self-repair in hardware requires resources (fault-detection logic, self-replication mechanisms, ...) that are normally not available in off-the-shelf circuits (typically, FPGAs). For these reasons, over the past several years a number of dedicated hardware devices have been developed, e.g. [6][12][14][15].

These devices have been successfully used to explore various bio-inspired paradigms, but in general they represent experimental platforms that are very difficult to program and require an in-depth understanding of the underlying hardware. As a consequence, these platforms are accessible only to a limited class of programmers who are well-versed in hardware description languages (such as VHDL) and who are willing to invest considerable time in learning how to design hardware for a specific, often ill-documented device.

Notwithstanding these issues, hardware remains an interesting option in this research domain as it can greatly accelerate some operations and because it allows a direct interaction with the environment. It is however undeniable that the difficulty of efficiently programming hardware platforms has prevented their use

for complex real-world applications. In turn, the fact that experiments have been mostly limited to simple demonstrators has hindered the widespread acceptance of the bio-inspired techniques they were meant to illustrate.

While, by their very nature, bio-inspired systems rely on non-conventional mechanisms, in the majority of cases they bear some degree of similarity to networks of computational nodes, a structure that is frequently used when dealing with parallel computing systems. In these systems, a software abstraction is often used to hide the complexity and details of the underlying hardware and simplify programming. It is therefore licit to wonder if such an approach can be applied to bio-inspired hardware systems as well, to allow researchers to rapidly prototype new ideas and, more importantly, to cope with the complexity of tens or hundreds of parallel computational elements.

In this article, we describe a set of tools that attempt to tackle this problem by providing a complete hardware and software design environment for distributed cellular architectures. More precisely, we will present a hardware platform that is able to support the necessary networking and computing elements that can then form what we call a scalable *Network-Of-Chips*. Built atop that hardware system lies a software framework that provides a model of the system to simplify its programming. Thus, it becomes possible to quickly evaluate new algorithms, techniques and ideas in the field of bio-inspired computing but also to harvest more easily the computing resources of hundreds of FPGAs, a number that can be smoothly scaled to higher figures thanks to a design based on the *GALS* (Globally asynchronous, locally synchronous) synchronization paradigm [4][5].

The structure of this article is as follows: in the next section, we present a brief overview of previous work in the domain. This will lead us to the presentation of the basic computational element of our system, a processor that can easily implement the kind of mechanisms required for bio-inspired applications but that remains sufficiently general-purpose to be compatible with conventional compilers. In chapter 4.2 we will detail the hardware implementation of our network layer before explaining, in chapter 4.3, how it is supported by the software framework. Before concluding, we will present how the whole framework can be used to develop a distributed cellular automaton and extended to less regular computations.

1.1 Background

The complexity of living organisms is based on multi-cellular organization where cells having a limited function achieve very complex behaviors by self-assembling into specific structures and operating in parallel. By analogy, we try in our approach to mimic this organization by replicating similar, relatively simple computing elements that can self-organize and execute in parallel the different parts of a given application.

To bridge the gap between programmable logic and the kind of software tools required to implement real-world applications, we opted for processor-scale computational elements that provide an environment for running a thread of a distributed application. These elements also meet quite closely the requirements

of our bio-inspired computing approach: substantially different from conventional computing units, these processors [13] possess some key features, described in some detail in section 3.2, that make them well-suited to implement the cells of our multi-cellular organisms.

On the hardware side, the platform we used for our experiment is Confetti (for *CONFigurable ElecTronic TIssue*) [9]. This platform consists of a scalable three-dimensional array of FPGAs that provides a considerable amount of computational resources and greatly enhances the communication capabilities of the hardware setup, compared to traditional solutions. As a consequence Confetti (described in section 2) represents an ideal platform for the kind of systems we are targeting. For instance, they allow the implementation of arbitrary connection networks for inter-processor communication, an invaluable capability both to approximate the kind of highly-complex communication that allows biological cells to exchange information within an organism and to instantiate *self-organization algorithms*, i.e., techniques that allow the system to organize its topology according to application requirements.

On the software side, we have been developing a *design flow* that leads from application code, written in C, to a complete parallel system implemented on a hardware substrate. This design flow includes a hardware-software partitioner [10] that helps the user determine how to optimize the processor for the application as well as a GCC back-end to generate the code to be executed.

In the context of the design of bio-inspired hardware systems, two aspects of these tools are particularly useful. First, they are designed in such a way that it becomes relatively simple to introduce mechanisms such as learning, evolution, and development to any application. Indeed, we have shown how evolution can play an important role in the design flow [10]. The second useful feature of the tools is that they are not, for the most part, tied to a hardware implementation: while we use the above-mentioned hardware setup in our experiments, most of the tools are quite general and can be applied to almost any network of computational nodes, whether they be conventional processors or dedicated elements. This flexibility comes from a decoupling between the hardware and the software layers, which allows the programmer to prototype bio-inspired approaches without necessarily knowing hardware description languages and specific implementation details.

2 The Confetti hardware platform for cellular computing

The Confetti platform, presented in [9], is composed of a set of stacks of printed-circuit boards (PCBs) that can be linked together side by side to form computational arrays of arbitrary size. As shown in Fig. 1, each stack is itself composed of four kinds of boards :

- The topmost layer of the stack consists of a 48x24 LED display with 18 touch-sensitive areas;
- All power supplies required by the system are handled by a second board that also handles functions such as startup and monitoring;

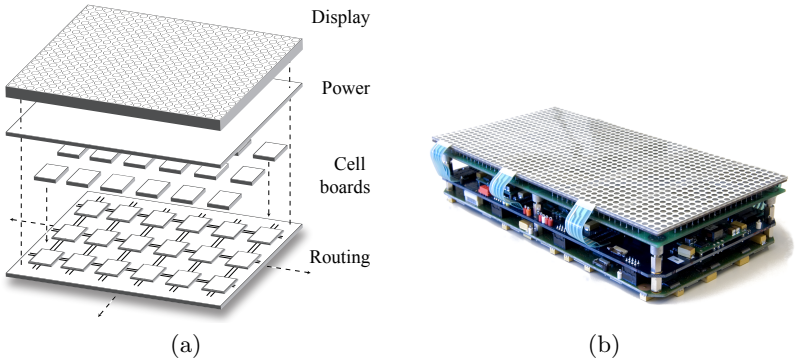

Fig. 1. A complete stack schematic (a) and photography (b).

- The *Routing* board implements the communication layer of the system. Articulated around eighteen dedicated FPGAs, the board implements a routing network based on a mesh topology which provides inter-FPGA communication as well as communication to other routing boards;
- The *Cell* boards (up to eighteen per stack) represent the computational part of the system and are composed of an FPGA and memory. Each *Cell* can be directly plugged into a corresponding routing FPGA in the subjacent *Routing* board.

A complete CONFETTI system consists of an arbitrary number of these stacks seamlessly joined together using border connections available in the *Routing* board. Connecting several stacks together potentially allows the creation of an arbitrarily large surface of programmable logic that is used, in this present work, as a network of CPUs where all components can be reconfigured.

This hardware structure proposes an increased amount of versatility compared to other platforms (for example, the BioWall [12] or the POEtic tissue [14]), notably because its modular organization allows interchanging all the elements of the system. If this might be interesting in the perspective of debugging and replacing faulty parts, the clear advantage of this approach resides in the fact that the computing elements, which are plugged into the system and not soldered on it, could also be easily replaced. This latter option is of particular interest in the larger perspective of a prototyping board for unconventional computing: nothing prevents the replacement of the current cells with more "exotic" or non-standard units that could potentially be of interest for research in bio-inspired mechanisms.

3 Computing architecture

The CONFETTI platform provides a powerful hardware substrate for the implementation of cellular systems and this section describes how it can be used to

implement the processing elements of our approach. Starting with the physical level, we will then examine the computational level formed by a specialized processor and, finally, the software support as seen by the programmer.

3.1 Physical layer - The *Cell* board

These small PCBs host a Xilinx Spartan 3 XC3S200 FPGA coupled with 8 Mbits of SRAM memory, resources that allow the implementation of relatively complex logic designs and, most notably, host a rather powerful processor. Of course, the presence of the FPGA implies that the structure of the cells is completely reconfigurable, allowing the definition of application-specific processing elements.

3.2 Computational layer - The Ulysse processor

To exploit the reconfigurability offered by the *Cell* board, we used the FPGA to implement a processor that was sufficiently generic to have its code generated by a compiler whilst maintaining the necessary amount of reconfigurability and flexibility to fulfill the different roles it could have in the context of bio-inspiration. Notably, our approach implies that it must be possible to *adapt the structure of the processor to the application* (as described in [13]). To achieve this goal, we have exploited a little-known approach known as the *Move* paradigm to implement our processor, called Ulysse.

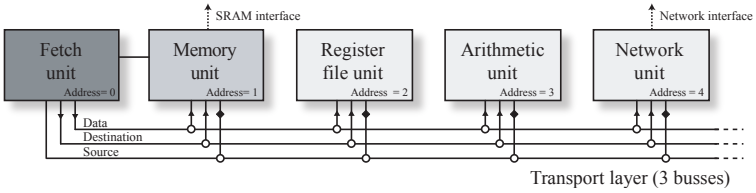

Fig. 2. Internal structure of the Ulysse processor

Belonging to the class of transport-triggered architectures (TTA, see [3]) this processor, rather than being structured, as is usual, around a more or less serial pipeline, relies on a set of *functional units* (FUs) connected together by one or more *transport busses* (see Fig. 2). All computation is carried out by the functional units (examples of such units can be adders, multipliers, register files, etc.) and the role of the instructions is simply to move data to and from the FUs in the order required to implement the desired operations. Since all the functional units are uniformly accessed through input and output registers, instruction decoding is reduced to its simplest expression, as only one instruction is needed: `move`. This approach, in and for itself, does not imply high performance, but several arguments in favor of TTAs have been proposed [3][7], the most important, in

the context of this article, is that new instructions can be added easily in the form of new FUs.

This architectural flexibility is all the more valuable because the processor is implemented in programmable logic and therefore exists as a VHDL description, easily parametrizable (e.g., to handle different data widths) and modifiable (e.g., to include a variety of more or less complex functional units). In the implementation used here, for example, the Ulysse processor uses 32-bit wide data and can be configured (by changing a simple value and re-synthesizing it) with optional modules such as a multiplier, a divider, a hardware timer, ...

On the performance side, the Ulysse processor attains 60 MIPS when only internal memory is used for data and instructions. When the external SRAM memory is used (which is normally the case due to the limited internal memory available in the FPGA), the performance drops to about 13 MIPS because the memory chips used imply three-cycle memory load operations and two-cycle store operations. The implementation of caching techniques or the use of a different kind of memory on the *Cell* board would have a major impact on performance, which in any case remains sufficient for most applications (particularly since the execution of the application will be divided among an array of many processors operating in parallel).

3.3 Software layer - The software model

To take into account the requirements of real-world applications, it was necessary to be able to connect our custom processors to a conventional design flow. In particular, because one of our objectives resides in simplifying the tool-chain traditionally used when designing distributed cellular applications, a back-end for the GCC compiler has been created for the Ulysse processor. This setup provides a solid foundation supporting all the constructs of the C language and allows to run, within the reasonable limits imposed by the memory size and the hardware platform, almost any program.

Even if the size of this article does not allow a detailed description of the structure of the GCC back-end, some aspects of its development are worth mentioning to illustrate the difficulties introduced by the use of non-conventional processors. In fact, because GCC was made at first for standard RISC/CISC processors in which operations are not simple displacements, it was necessary to tell the compiler that moves into the trigger registers had to be seen as the operations themselves. Moreover, the fact that some registers are *read-only*, such as the registers holding results of operations, also created problems because every register is normally expected to be fully accessible by the compiler when performing some of the optimization passes. Furthermore, due to the number of constraints on each register in the processor, GCC has sometimes trouble finding solutions for register allocation.

Of course, due to the complexity of porting such a compiler, a few problems still exist today. First, some optimizations are not always possible, a situation that limits the efficiency of the code. Secondly, neither software floating-point operations nor the C standard library have been entirely ported yet. Still, except

for these limitations, the compiler is fully working and the results obtained so far are very encouraging: they open the way to the realization of high-complexity applications on our bio-inspired substrate.

4 The communication infrastructure

One of the main challenges in today's hardware architectures resides in implementing versatile communication capabilities that are able to provide a sufficient bandwidth whilst remaining cost- and size-efficient, as evidenced in research on *Network-On-Chip* [2] and other systems [8]. This aspect of parallel systems is unfortunately often ignored in bio-inspired hardware approaches and is another of the factors that prevent the implementation of complex applications. In our case, the addition of a network layer enables data to be moved not only within the various FUs of the processor but also outside of it to realize a complex *Network-Of-Chips* of virtually unlimited size.

To accomplish this, the functions of computation and communication were logically and physically split. Thus, we were able to conserve the whole computational resources of the *Cell* board whilst staying very flexible for the communication network in our system, implemented within the *Routing* board that shall be now described.

4.1 Physical layer - The *Routing* board

The physical layer of the network infrastructure is built using a six-by-three regular grid of FPGAs on the top of which the same number of *Cell* boards can be plugged. The use of reconfigurable circuits at the routing level enables the exploration of various algorithms and techniques for data transport in the system.

As the *Routing* boards constitute the communication backplane of the Confetti platform, connections between the different boards are also implemented here. External connectors on the four sides of the board provide the same connectivity as the links between the FPGAs: two adjacent *Routing* boards then effectively represent a single uniform surface of FPGAs. This setup allows the creation of systems consisting of several stacks that behave as a single, larger stack (at the time of writing, six of these stacks have been built).

4.2 Transport layer - The Mercury interface

Since scalability is a key feature of our architecture, no global clock is available to the system. This implies that synchronization between the different FPGAs is not straightforward, notably to transmit information.

Since transmission delays can be relatively long in our system, the usage of a *request-acknowledge pair* as traditionally used in asynchronous transmission would be too slow (early tests showed that the maximum bandwidth using that technique was limited to about 50 Mbits per second). Thus, we chose to use a

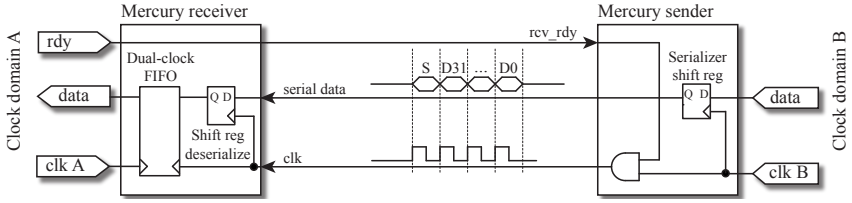

Fig. 3. Simplified schematic of the MERCURY basic units.

method that transmits the clock along with the data. The steps of a transmission are as follows: the receiver generates an acknowledge signal when ready to get data and the sender can then transmit the data synchronously with the clock signal it generates, starting with a start bit that indicates the beginning of the data. This solution allows completely unrelated clocks to be used in each clock domain and uses only three lines per link (using the FPGA differential I/O drivers) to implement full-duplex transmission in each direction.

This basic interface is encapsulated into a bigger unit that, on one hand, serializes and de-serializes the data to form 32-bit words and, on the other hand, allows to operate at different speeds thanks to dual-clock FIFOs also used to interface to the Ulysse CPU. Thanks to this transmission module, called MERCURY, a bandwidth of approximately 200 Mbits per second in full-duplex was obtained.

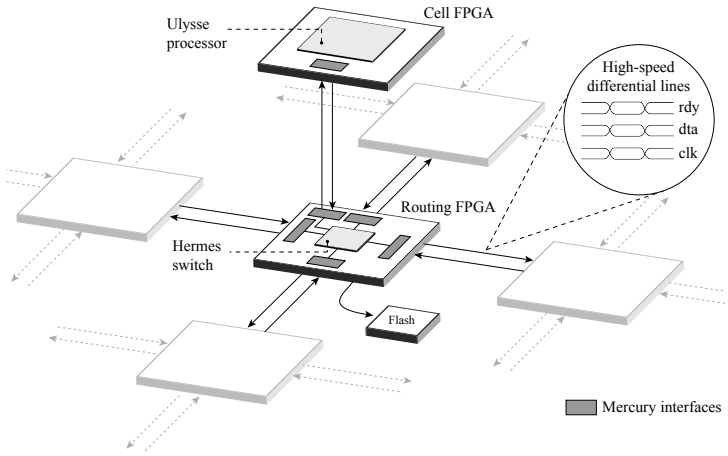

Fig. 4. Detail of one *Routing* FPGA and link with its *Cell* module.

Physically, every FPGA on the routing substrate is linked to its four cardinal neighbors and to the *Cell* board above it (Fig. 4) using MERCURY interfaces. This mesh topology was selected for its modularity and scalability (it avoids long and

global communication lines) and because it is the kind of layout typically used in cellular computing applications. Of course, many different types of networking paradigms exist and could be implemented in our system (for example [11], [16] or [1]).

To route the data between the different MERCURY interfaces, we used the switch-box element of the *Hermes* framework [8] in a slightly modified version to handle serial data transmissions. Its role is to redirect packets, using the addresses embedded in the data headers, that could come from any direction to any other direction, enabling point-to-point communication between *any* arbitrary pair of *Cell* boards in the whole CONFETTI system.

4.3 Software layer - The messaging layer

Following the layered approach traditionally used in communication protocols, we implemented on top of the MERCURY physical layer a software *transport layer* that manages messages. The role of this component is to handle *packet* transmission and reception, providing more complex communication schemes than simple point-to-point communication. It consists of two essential parts: the first acts as a driver for MERCURY proper and provides simple send/receive functionality. Using it, the second part implements useful operations such as *broadcasting* or *guaranteed-delivery packets*. It also allows waiting for, then receiving, a packet fulfilling a set of given criteria.

In practice, this layer extends MERCURY data packets into its own format to include sender, message type and sequence number information. It is worth noting here that the CPU can be reprogrammed by sending a packet with a reserved message type that is intercepted in hardware. Its payload replaces then the previous program code, enabling a very convenient and fast way to dynamically update processor code. This feature provides not only an easy way to upload code to a CPU from a host machine (where the code is compiled) but also allows interesting behaviors: for instance, the API provides a function to replicate a CPU to another position in the network and another to replicate a given CPU across the entire grid, two examples that illustrate how the hardware-software synergy is used in the developed framework.

5 An application example: CAFCA

In this section, we will validate our hardware–software design approach by showing how a traditional cellular automaton can be modeled within our framework. The task is not trivial because of the lack of global synchronization clocks in the CONFETTI system. Typically, this absence would imply a considerable effort on the programmer's part to design and implement alternative synchronization mechanisms, again requiring a strong knowledge of VHDL and of the detailed operation of the hardware.

To prevent this limitation, we developed CAFCA (*Cellular Automata Framework for Cellular Architectures*), a software library that runs on the Ulysse processor and that essentially permits, thanks to the messaging layer, to virtualize

the fact that we are running a globally asynchronous system. This tool represents a good example of the kind of software interfaces that can allow virtually any researcher to exploit the power of bio-inspired hardware implementations. In fact, when using this library it becomes then possible to develop parallel applications based on the cellular automata model by writing only a few C functions without any particular knowledge of the hardware. Because the development of the library had to be done only once, it greatly enhances the usability of the system.

To achieve a synchronized behavior, a step in the cellular automaton model implemented by CAFCA consists of four phases: synchronization, inter-neighbor state exchange, next-state computation and display. CAFCA builds upon the messaging layer and leverages the grid topology of the underlying hardware to provide cell synchronization and state exchange between neighbors that belong to different clock domains. Technically speaking, this is implemented with one of the cells playing the role of a supervisor to ensure, using broadcast messages, that every cell is at the same step.

An application that uses the framework library needs to provide only the functions for the computation and display phases, as well as emplacements to store local/neighbor states and their size, as shown on Fig. 5. By fully handling synchronization and messaging issues, CAFCA enables fast development strictly centered on the two defining characteristics of a cellular automaton – the state variables and the state update function. As an example, the implementation of Conway's Game of Life in our framework takes less than 100 lines of C code (corresponding to the user code in Fig. 5). Other examples we developed, such as a simplified version of heat distribution in homogeneous metal or the simulation of shallow water equations, can be coded in less than one thousand lines.

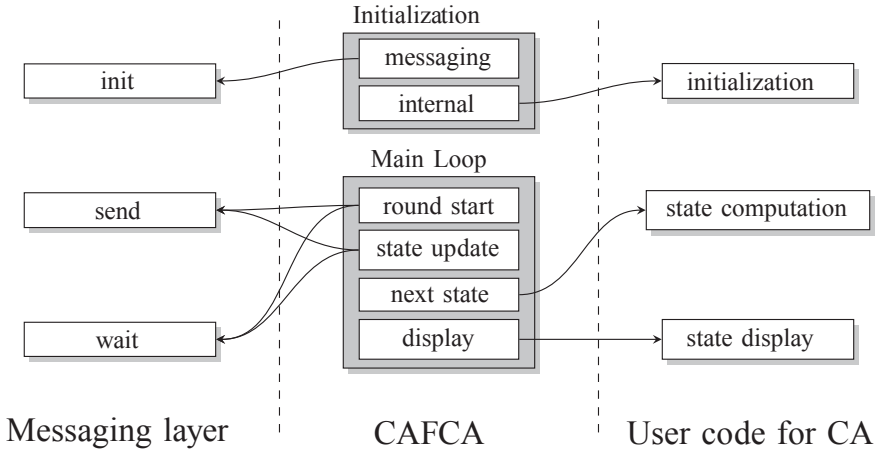

Fig. 5. Structure of an application using CAFCA. Arrows represent function calls.

6 Conclusions and future work

In order for bio-inspired concepts to be accepted as valuable tools for the design and use of digital computing systems, they will have to be tested and verified on complex real-world applications. The experimental setups associated with research in this field, however, are often of limited use in this context because they remain either too specific or too difficult to use beyond the proof-of-concept stage.

When examined in comparison with conventional approaches used to design complex systems, bio-inspired design is not so much limited by the hardware (which, on the contrary, is often quite innovative and powerful) but rather by the lack of the kind of software tools that are at the core of any industrial-strength digital design flow.

As an example of the type of tools that could be invaluable for research in bio-inspired hardware, we showed in this article how the framework we have developed can be used to provide an easy and rapid way to implement a traditional task for cellular hardware systems in the form of cellular automata. By developing several hardware and software components, we were able to drastically reduce the non-recurrent engineering time generally implied by the usage of intimately linked hardware and software by delegating all the complexity to a framework which presents to the programmer a simplified, yet powerful, view of the system. Thus, with a tool like CAFCA, every programmer that knows C could exploit the computational power of hundreds of FPGAs with just a few lines of standard code. The development of other software libraries could, for example, take advantage of the reconfigurability capabilities of the system to implement growth or replication but also to realize computationally intensive functions such as video decoding by implementing *ad-hoc* FUs.

This work was done in the context of the development of a complete automated software suite for bio-inspired systems generation. The task is complicated by the dynamic operations that characterizes biological systems: the structural adaptation implicit in processes such as development, learning, and evolution requires reconfigurability to be exploited to an unprecedented degree. The approach we propose takes advantage of reconfigurability on several levels: at the design level, where the VHDL description of the processors and of the network can be manipulated through a set of automated software tools, at the system level, where processing nodes can be spawned and killed to satisfy the needs of the application or to respond to faults, and at the processor level, where the processor can self-reconfigure both its executable code and its very structure (by reconfiguring the FPGA that implements its functional units).

Our final objective is then to develop a set of design tools that will allow researchers to exploit this reconfigurability easily and without a specific knowledge of the underlying hardware in order to harness the power of bio-inspired approaches in the context of real-world applications.

References

1. M. Amde, T. Felicijan, A. Efthymiou, D. Edwards, and L. Lavagno. Asynchronous On-Chip Networks. *IEE Proceedings Computers and Digital Techniques*, 152(02), March 2005.
2. T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.*, 38(1):1, 2006.
3. H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, NY, USA, 1997.
4. W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnectoin networks. In *DAC '01: Proc. 38th Conf. on Design automation*, pages 684–689, New York, NY, USA, 2001. ACM Press.
5. G. de Micheli and L. Benini. Networks on chip: A new paradigm for systems on chip design. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 418, Washington, DC, USA, 2002. IEEE Computer Society.
6. A. Greensted and A. Tyrrell. RISA: A hardware platform for evolutionary design. In *Proc. IEEE Workshop on Evolvable and Adaptive Hardware (WEAH07)*, pages 1–7, Honolulu, Hawaii, April 2007.
7. J. Hoogerbrugge and H. Corporaal. Transport-triggering vs. operation-triggering. In *Proc. 5th Intl. Conf. on Compiler Construction*, pages 435–449, 1994.
8. F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integrated VLSI Journal*, 38(1):69–93, 2004.
9. P.-A. Mudry, F. Vannel, G. Tempesti, and D. Mange. Confetti : A reconfigurable hardware platform for prototyping cellular architectures. In *Proc. 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS07)*, page 186, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
10. P.-A. Mudry, G. Zufferey, and G. Tempesti. A Dynamically Constrained Genetic Algorithm For Hardware-software Partitioning. In *Proc. of the 8th annual conf. on Genetic and evolutionary computation GECCO'06*, pages 769–776, Seattle, 2006.
11. A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, and A. Susin. A contextual resources use: a proof of concept through the APACHES' platform. In *Proceedings of the 2006 IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 44–49, April 2006.
12. G. Tempesti, D. Mange, A. Stauffer, and C. Teuscher. The BioWall: an electronic tissue for prototyping bio-inspired systems. In *Proc. 3rd Nasa/DoD Workshop on Evolvable Hardware*. IEEE Computer Society.
13. G. Tempesti, P.-A. Mudry, and R. Hoffmann. A Move Processor for Bio-Inspired Systems. In *Proc. NASA/DoD Conf. on Evolvable Hardware (EH2005)*, IEEE Computer Society Press, pages 262–271, 2005.
14. Y. Thoma, G. Tempesti, E. Sanchez, and J.-M. Moreno Arostegui. POEtic: An electronic tissue for bio-inspired cellular applications. *BioSystems*, 74(1-3):191–200, Aug.-Oct. 2004.
15. A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas. The perplexus bio-inspired reconfigurable circuit. In *Proc. 2nd NASA/ESA Conf. on Adaptive Hardware and Systems (AHS07)*, pages 600–605, Washington, DC, USA, 2007. IEEE Computer Society.
16. D. Wiklund and D. Liu. SoCBUS: Switched network on chip for hard real time embedded systems. In *IPDPS'03: Proc. 17th Intl. Symposium on Parallel and Distributed Processing*, page 78.1. IEEE Computer Society, 2003.