A *Move* Processor for Bio-Inspired Systems

Gianluca Tempesti, Pierre-André Mudry, and Ralph Hoffmann

Logic Systems Laboratory
Swiss Federal Institute of Technology at Lausanne (EPFL)
EPFL-IC-LSL, INN Ecublens, CH-1015 Lausanne, Switzerland
Email: Gianluca.Tempesti@epfl.ch

Abstract. The structure and operation of multi-cellular organisms relies, among other things, on the specialization of the cells' physical structure to a finite set of specific operations. If we wish to make the analogy between a biological cell and a digital processor, we should note that nature's approach to parallel processing is subtly different from conventional von Neumann architectures or even from conventional parallel processing approaches, where specialization is obtained by adapting software to a fixed hardware structure.
In this article we will present the outline of a novel processor architecture based on the Move or TTA (Transport-Triggered Architecture) approach. The features of such architectures allow them to implement systems that more closely resemble, within the limitations imposed by the capabilities of conventional silicon, the general *modus operandi* of multi-cellular organisms.

1 Introduction

One of the main motivations for the development of hardware-based bio-inspired systems is the astounding level of complexity achieved by biological organisms, a complexity far beyond that of even the latest silicon-based circuit. The promise of next-generation technologies [1][2][3] lies in their ability to work at the same molecular level, with comparable component densities, as biological systems.

Among the many questions open for these technologies is how to exploit this immense wealth of hardware. The study of how biology, and notably multi-cellular organisms, have successfully solved this issue is a possible avenue for finding approaches that could potentially be applied to these circuits.

Of particular interest in this context is the biological process of *ontogenesis*, whereby molecules self-assemble into cells and cells self-assemble into complete organisms, according to a (very compact) set of instructions contained in the genome. A possible analogy between biological systems and electronics is to compare a cell to a digital processor, implying a correspondence between an organism and a massively parallel multi-processor system. This analogy holds in several respects, but it should be noted that nature's approach to parallel processing is subtly different from conventional von Neumann architectures or even from conventional parallel processing approaches, where specialization is obtained by adapting software to a fixed hardware structure.

This article describes the first results of a new project that, building on the bases provided by the Embryonics [4] and POEtic [5] projects, will define a processor architecture specifically conceived for the realization of this kind of bio-inspired systems. In this paper, we will try to identify some of the requirements of an ontogenetic processor architecture and present the outline of a novel architecture that represents an effort towards the designs of systems that more closely resemble, within the limitations imposed by the capabilities of conventional silicon, the general *modus operandi* of multi-cellular organisms.

2 Background

Many different approaches can be used to draw inspiration from nature in the design of electronic systems. Even within the much more restricted area of ontogenetic hardware (that is, hardware inspired by the ontogenetic development of multi-cellular organisms), several valid approaches have been studied (for a partial review of such systems, see [6]).

Within the Embryonics project [4], we have been studying the application of biological ontogenesis to the design of digital hardware for several years. Among what we feel are our main contributions to the field is a self-contained representation of a possible mapping between the world of multi-cellular organisms in biology and the world of digital hardware systems (Fig. 1), based on 4 levels of complexity, ranging from the population of organisms to the molecule.

We define an artificial organism as a parallel array of cells, where each cell is a simple processor that contains the description of the operation of every other cell in the organism in the form of a program (the *genome*). The redundancy inherent in this approach is compensated by the added capabilities of the system, such as growth [7] and self-repair [8].

The structure and operation of multi-cellular organisms relies, among other things, on the specialization of the cells to a finite set of specific operations, implying that the cells' *physical structure* is adapted to its function (e.g., a skin cell is physically different from a liver cell). Structural differences notwithstanding, the same program (genome) controls the operation of all cells. To maintain the analogy with digital processors, we must achieve a similar degree of adaptation.

A first answer to this issue was to redefine our cells as *reconfigurable* processing elements, realized by programmable logic circuits and structurally adapted to the application to be implemented by the system. For a given application, all cells are structurally identical and contain the same program (and can thus be seen as *stem cells* [9]), but different parts of the program and of the structure are activated depending on the cell's position, implementing specialization.

In 2001, we launched, together with the universities of York, Barcelona (UPC), Lausanne, and Glasgow, a project called "Reconfigurable POEtic Tissue" [5] funded by the Future and Emerging Technologies programme (IST-FET) for the European Community. This project aims at defining a novel programmable circuit specifically designed for the implementation of bio-inspired systems and thus at providing an efficient molecular level for our systems.

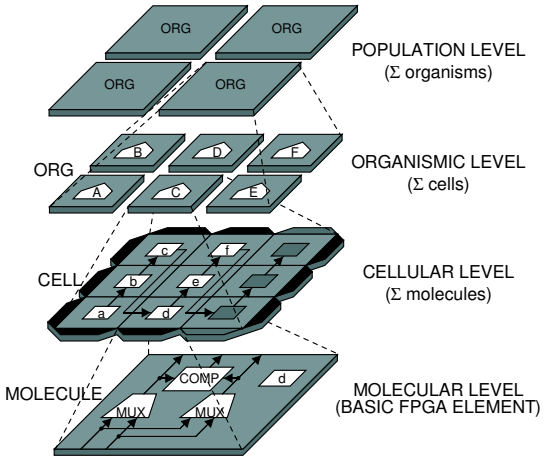

Fig. 1. The four hierarchical levels of complexity of the Embryonics project

3 Ontogenetic Processor Architectures

Exploiting the results of the projects mentioned above, we have begun to address some of the issues related to the implementation of the *cellular* level of our systems by defining some of the practical requirements of ontogenetic applications. We have then begun to work on developing a candidate solution in the form of a processor based on the *Move* approach.

3.1 Ontogenetic Applications

It is not simple to identify applications that can exploit the features of an ontogenetic approach on conventional silicon. A developmental process is likely to become a necessity for the next generation of molecular-scale circuits, but today's technology remains at a level of complexity that can be handled by more conventional design approaches. However, there exist some families of applications where ontogenesis can be useful today (see [6] for a more complete review).

A first set of applications can exploit structural adaptation to respond to environmental stimuli that cannot be foreseen at design time. Typically, these applications, which *self-organize around external stimuli*, correspond to other kinds of bio-inspired approaches, such as neural networks [10] or robotics [11], but the approach can be extended to applications where the circuit's function is determined by the user at runtime (e.g., custom graphic or sound filters [12]).

A noteworthy special case in this context are systems that exploit developmental processes for their capability to represent structural information in a compact form. This compactness is a major advantage when applying evolutionary approaches to hardware design. In this case, the fitness of the individuals can be seen as the external input around which the system is structured and the final individual cannot, by definition, be determined at design time.

A second, more general set of applications (not sufficiently exploited in the context of bio-inspired systems) could use development for the creation of massively parallel arrays of reconfigurable processors. The (relative) decline of massively multi-processor systems is usually explained by the difficulty of exploiting the parallelism inherent in many algorithms. In turn, it could be argued that at least part of this complexity lies in the *implementation* of these algorithms, usually written in a high-level language with a general-purpose instruction set.

A well-known technique to simplify the realization of algorithms on a massively parallel system is the use of *application-specific processors*: if the processing elements in the system are designed to execute a single application (or set of applications), the instruction set of the processor can be targeted to the required operations, leading to programs that are much simpler than those written for general-purpose processors. This approach can simplify the task of programming parallel systems by moving some of the software's complexity to the hardware.

The kind of ontogenetic systems we have been working on are ideally suited for this kind of approach: not only the processing elements are fully configurable (and can thus be made application-specific), but our developmental mechanisms allow the inter-processor communication network to adapt itself at runtime, letting the system *self-organize around the data flow*.

As an important special case of this kind of systems, we shall mention *fault-tolerant* processor arrays. The possibility of operating in the presence of hardware faults is not only a key feature of molecular-level computing [13], but also increasingly important for silicon-based circuits (error rates increase with the shrink in transistor size). In particular, in the Embryonics project we have been concentrating on a specific approach to fault tolerance: *on-line self-test and self-repair*. In this approach, the system must be able not only to detect that a fault has occurred in the hardware substrate, but also to self-repair (through reconfiguration) and to resume operation without losing its current state of operation.

This kind of fault tolerance is normally considered prohibitively expensive for commercial purposes because of its inherent overhead. However, ontogenetic systems are ideally suited for this kind of approach, as many of the mechanisms involved in the reconfiguration of the system following the detection of a fault are very similar to those required for the growth of a system. This property is an immediate consequence of the biological inspiration of our systems: in nature, self-repair (e.g., cicatrization) relies on the creation of new cells to replace those damaged by an illness or a wound, and the cellular division involved in this process is very similar to that used during the growth of the organism.

3.2 A MOVE Architecture

The requirements of our bio-inspired approach imply then an architecture that is substantially different from conventional general-purpose processor architectures: it must be possible to *adapt the structure of the processors to the application* to exploit the programmability of application-specific systems and it must be possible to *adapt the topology of the system to the application* to take advantage of the features of the ontogenetic approach.

To achieve this kind of adaptation within an array of processors, we exploited a relatively little-known approach, known as the *Move* or TTA (Transport-Triggered Architecture) paradigm [14][15][16].

In many respects, the overall structure of a TTA-based system is fairly conventional (which is an advantage as far as system design is concerned): data and instructions can be fetched to the processor from main memory using standard mechanisms (caches, memory management units, etc.) and are decoded within the processor as in conventional processors. The basic differences lay in the architecture of the processor itself, and hence in the instruction set.

Rather than being structured, as is usual, around a more or less serial pipeline, a *Move* processor (Fig. 2) relies on a set of *functional units* (FUs) connected together by a *transport layer*. 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, only one instruction is needed: `move`.

TTA `move` instructions trigger operations which in fact correspond to normal RISC instructions. So, for example, to add two numbers the processor would use a functional unit that implements the `add` operation, move one operand into the first input register of the unit, move the other operand into the second input, and move the result from the output register to the unit that needs it.

The *Move* approach, in and of itself, does not imply high performance: a simple addition, in our example, requires three move instructions. Its strength lies in its *modularity*: the architecture handles the functional units as "black boxes", without any inherent knowledge of their functionality. This property implies that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the address of the corresponding functional units. As we will show, this feature allows us to adapt the structure of the processors to the application by specializing the instruction set (i.e., the functional units) to the application while keeping the overall structure of the processor (fetch and decode unit, bus structure, etc.) and the syntax of the language (based on the single instruction `move`) unchanged.

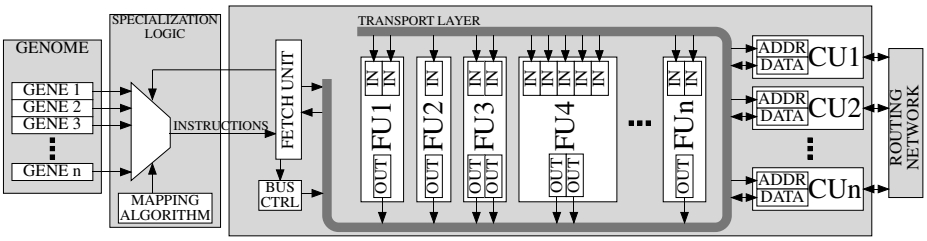

Fig. 2. A MOVE processor consists of a set of functional and communication units tied together by one or more data busses.

Moreover, communication channels can be handled exactly in the same way as a functional unit (Fig. 2): the address of the target cell can be moved to a dedicated input register in a communication unit (CU), and the data to a second input register. The unit can then autonomously set up the communication channel and transmit (or receive) the data. This key feature of the TTA approach implies that the connection network can be arbitrarily complex, as it is handled by the CUs without directly affecting the structure of the processor itself, and opens the way to the use of complex routing algorithms (e.g., dynamic routing networks [17]) that allow us to adapt the structure of the array to the application.

4 A Prototype System

For the implementation of ontogenetic systems, one of the key features of a TTA processor is therefore the possibility to easily parametrize its structure. The fetch and decode subsystems, the transport layer (that is, the busses that implement the datapath) and the functional units can each be modified more or less independently to fit the requirements of the application.

To test the flexibility of this approach, we realized a prototype system to experiment with a possible implementation for each of these subsystems. Our implementation choices represent a fixed compromise between performance and size, but we designed the system so that the specific parameters used can be very easily adapted to shift the balance one way or another.

4.1 Fetch and Decode

The processor fetch and decode cycle is relatively standard: the code to be executed is stored locally in a small memory and at every clock cycle the instruction pointed by the program counter (PC) is loaded and decoded. According to the TTA approach, there exists only a single instruction (`move`) with two formats:

$$\mathsf{Address} \rightarrow \mathsf{Address} \ : 0 \mid \underbrace{DDDDDDDD}_{\text{8-bit dest.}} \mid \underbrace{SSSSSSSS}_{\text{8-bit source}} 0 \tag{1}$$

$$\mathsf{Immediate} \rightarrow \mathsf{Address} \ : 1 \mid \underbrace{DDDDDDDD}_{\text{8-bit dest.}} \mid \underbrace{IIIIIIIII}_{\text{9-bit imm. value}} \tag{2}$$

As a test of the parametrization capabilities of the approach, we adopted a VLIW (Very Long Instruction Word) encoding for our instructions, allowing our processor to execute two `move` instructions in parallel. Every 36-bit *instruction word* can then contain up to two of the above instructions.

After a short decoding phase where eventual immediate values are extracted from the instruction, the *Fetch Unit* sends the source and destination addresses of the registers to the functional units through the transport layer. By permanently scanning the busses, the functional units can then know if they are involved the operation either as a source or a destination for the data transfer.

As each address in the instruction uniquely identifies an I/O register of a functional unit, the format of the instructions imposes some upper limits on the size of the processor (in this case, an instruction can address up to 256 source or destination registers). However, the size of the instructions, and hence the size of the processor, can be altered easily since the decode logic is extremely simple.

4.2 The Transport Layer

In our implementation, we opted for a shared-bus topology for the transport layer: for each of the two instructions encoded in a word, three separate busses (one each for the 8-bit source and destination addresses and one for the 32-bit datum being moved) connect all the functional units of the processor.

4.3 The Functional Units

In the *Move* paradigm, the functional units define the instruction set of the processor by implementing the operations required by the application and by acting as sources or destinations of data displacements. This approach also implies that the instruction set can be easily modified by adding or removing functional units.

To implement this functionality, we have developed a common bus interface, used by every functional unit to connect to the transport layer. This interface lets heterogeneous components be accessed uniformly and allows the processor to be assembled using a library of pre-defined functional units (written in VHDL). We have then developed a small set of basic FUs, separated in three main classes:

1. *Computational Units*
   This class contains the classical arithmetic and logic operations found in a conventional processor. To this class also belong most of the application-dependent functional units that can be used to customize the processor. In our prototype, we included the basic operations: add, sub, multiply, shift, and some logical operations.
2. *Operational Units*
   This class contains the units required to control the processor, such as a *register file* containing a parameterizable number of general-purpose registers (eight 32-bit registers in our prototype), a *condition unit*, used for branching, offering several comparison schemes (the result of the comparison can then be moved to the fetch unit to serve as a condition for a *jump*), and a *data memory*, which corresponds to the data cache and to the data memory management unit and offers various addressing modes such as stack or auto-incremented addressing (a 512x32-bit memory in the prototype).
3. *I/O Units*
   This class is used to implement the network that connects the processors to each other. In our prototype, the I/O units are 32-bit registers used to implement a simple shared-bus topology that connects all processors in the system. However, as in the TTA approach outside communication is handled as a standard data displacement, these units can become arbitrarily complex.

4.4 Development Tools

To test the software-side implementation issues of the TTA paradigm, we wrote an assembler and a minimal simulation environment. These tools are *qualitatively* interesting, as we included some of the key elements required to efficiently exploit the features of transport-triggered architectures.

With an instruction set reduced to its simplest expression with only two variants of a single instruction, an assembler for a *Move* processor does not have to handle complex instructions encodings. However, the need to use only the `move` instruction makes programming a TTA processor considerably more difficult than a conventional one, since the level of abstraction usually provided by standard assembly languages is missing.

To overcome this problem, we have designed an assembler that offers an extendable set of *macro-instructions* used to define a "meta-language" that permits to express programming concepts more intuitively. In practice, sets of `move` instructions are grouped to form the instructions of the new meta-language (e.g., the `add` macro-instruction corresponds to a set of three `move` instructions).

Our meta-language contains all of the conventional RISC instructions (including abstractions such as conditional jumps, load and store instructions, or function calls). The assembler can use these macro-instructions in place of the primitive `move` instructions and thus allows out TTA processor to be programmed not unlike a conventional RISC processor.

4.5 The Memory Map

As we mentioned, the *Move* paradigm implies that every FU corresponds to an address range in a memory map. To design application-dependent implementations, the address map of the functional units is defined in a file which is accessed at assembly time. This file makes the relation between the physical address space and a set of symbolic address names used in assembly code. As a consequence, the physical units (i.e., the VHDL code) are separate from their software representation, implying that the assembled code is compatible across implementations that share a common subset of FUs.

Defining the instruction set in a file also simplifies the specialization of TTA processors: if a given algorithm would benefit from the use of a specific hardware function (e.g., a FFT), a custom FU can be designed and added to the memory map, where it could immediately be exploited by the assembler.

4.6 Implementation

To verify the implementation of our prototype, we instantiated into a Xilinx Virtex II-3000 FPGA a matrix of twelve processors running at approximately 50 MHz (Fig. 3). Each processor is independent from the others and runs its own program, uploaded dynamically from the host PC. As we mentioned, the interconnection network is very basic (a shared-bus architecture where all processors and the host PC, who initiates all transfers, are connected by a single bus) but sufficient for the purpose.

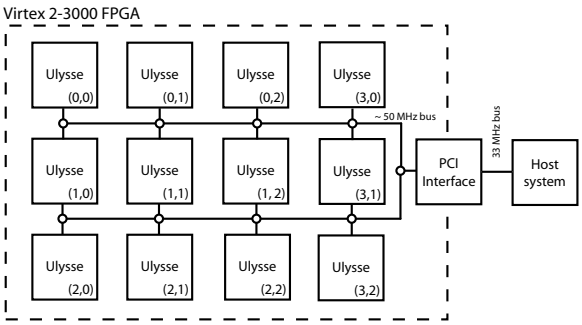

Fig. 3. Overview of the prototype system.

To test the functionality of the processors, we built a demonstration application that uses two of the twelve processors : the first computes the factorial of a number input by the user whereas the other computes a graphical *plasma* effect. The latter is computed by combining various trigonometric functions taking the time and position of each considered picture element as parameters. The result is then displayed on the host screen in a Java GUI where the user can input values for the factorial program and visualize the real-time plasma effect.

5 Conclusions

The implementation of ontogenetic systems in silicon using the approach defined in the Embryonics and POEtic projects amounts to the creation of massively parallel arrays of application-specific processors with properties, such as growth and self-repair, typical of biological organisms.

Two practical considerations stand in the way of such an implementation. The first is technological: current reconfigurable circuit densities do not allow the realization of massively parallel systems. However, improvements in silicon technology and, eventually, the development of molecular-level circuits should not only allow such systems to be built, but even *require* some of their features.

The second consideration concerns the implementation of ontogenetic systems: there exists today no universal architecture for application-specific processors that can be used to implement effectively our approach. The processor architecture presented in this article, coupled with some of the bio-inspired concepts developed for the Embryonics and POEtic projects (such as the presence of the genome in every cell, the use of a growth algorithm to control the topology of the network, and the presence of a dynamic routing network) responds to several of the necessary criteria for the implementation of ontogenetic systems and represents a step forward for the creation of systems that more closely resemble the organization and operation of multi-cellular systems in nature.

Future work within the project calls for two main axes of research. On one axis, we will pursue the development of the processor by investigating in detail

the different implementation parameters for the processor and by setting up a complete design environment to facilitate its use. On another axis, we will integrate to the architecture the most interesting features of bio-inspired systems, introducing high-level processes such as growth, learning, and fault-tolerance.

References

1. Drexler, K.E.: Nanosystems: Molecular Machinery, Manufacturing and Computation. John Wiley, New York, NY (1992)
2. Goddard III, W., Brenner, D., Lyshevski, S., Lafrate, G., eds.: Handbook of Nanoscience, Engineering, and Technology. CRC Press, Boca Raton, FL (2002)
3. Lent, C., Tougaw, P.: A device architecture for computing with quantum dots. Proceedings of the IEEE **85:4** (1997) 541–557
4. Mange, D., Sipper, M., Stauffer, A., Tempesti, G.: Towards robust integrated circuits: The embryonics approach. Proceedings of the IEEE **88** (2000) 516–541
5. Tyrrell, A., Sanchez, E., Floreano, D., Tempesti, G., Mange, D., Moreno, J.M., Rosenberg, J., Villa, A.: Poetic tissue: An integrated architecture for bio-inspired hardware. In: Proc. 5th Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES2003). Volume 2606 of LNCS., Springer Verlag (2003) 129–140
6. Tempesti, G., Mange, D., Petraglio, E., Stauffer, A., Thoma, Y.: Developmental processes in silicon: An engineering perspective. In: Proc. 2003 NASA/DoD Conference on Evolvable Hardware (EH-2003), IEEE Computer Society Press, Los Alamitos, CA (2003) 255–264
7. Mange, D., Stauffer, A., Petraglio, E., Tempesti, G.: Embryonic machines that divide and differentiate. In: Proc. 1st Int. Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT04). (2004)
8. Tempesti, G., Mange, D., Stauffer, A.: A robust multiplexer-based fpga inspired by biological systems. Journal of Systems Architecture **43** (1997) 719–733
9. Pearson, H.: The regeneration gap. Nature (2001) 388
10. Vaario, J., Onitsuka, A., Shimohara, K.: Formation of neural structures. In: Proc, 4th European Conference on Articial Life (ECAL97), MIT Press (1997) 214–223
11. Bongard, J., Pfeifer, R.: Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In: Proc, of the Genetic and Evolutionary Computation Conference (GECCO-2001), Morgan Kaufmann (2001) 829–836
12. Sekanina, L.: Virtual reconfigurable circuits for real-world applications of evolvable hardware. In: Proc. 5th Intl. Conf. on Evolvable Systems: From Biology to Hardware (ICES03). Volume 2606 of LNCS., Springer Verlag (2003) 186–197
13. Han, J., Jonker, P.: A system architecture solution for unreliable nanoelectronic devices. IEEE Transactions on Nanotechnology **1** (2002) 201–208
14. Corporaal, H.: Microprocessor Architectures – from VLIW to TTA. John Wiley & Sons (1998)
15. Corporaal, H., Mulder, H.: MOVE: A framework for high-performance processor design. In: Proceedings of the 1991 International Conference on Supercomputing. (1991) 692–701
16. Tabak, D., Lipovski, G.J.: MOVE architecture in digital controllers. IEEE Transactions on Computers **C-29** (1980) 180–190
17. Thoma, Y., Sanchez, E., Moreno Arostegui, J.M., Tempesti, G.: A dynamic routing algorithm for a bio-inspired reconfigurable circuit. In: Proc. 13th Int. Conf. on Field-Programmable Logic and Applications (FPL03). Volume 2778 of LNCS., Springer Verlag (2003) 681–690