Relation Between Fault Tolerance and Reconfiguration in Cellular Systems

Lukáš Sekanina and Vladimir Drábek
Department of Computer Science and Engineering
Brno University of Technology
Brno, Czech Republic
sekanina@dcse.fee.vutbr.cz, drabek@dcse.fee.vutbr.cz

Abstract

Recently, hardware researchers have promptly begun to investigate alternative computational principles to the conventional ones. The main signs of these principles are inspiration in biology and their direct hardware implementation. Evolvable hardware, cellular computing or embryonic electronics are the most important examples. This paper describes different approaches to configuration, reconfiguration and fault tolerance implementation of two-dimensional cellular system. Simplicity of the cell, vast parallelism, and the connection locality are considered as the design restrictions.

1. Introduction

New bio-inspired systems don't want to copy natural processes; nature is only a source of inspiration. Living beings exhibit such qualifications as evolution, adaptation and fault tolerance that have difficult implementation using traditional methodologies. *Phylogeny* and *ontogeny* [5] are nowadays ones of the most popular natural phenomena in hardware design. From the technological point of view, these experiments can be done in the field of digital circuit design, because the reconfigurable VLSI digital circuits (e.g. *Field Programmable Gate Arrays*) have been developed.

Phylogeny concerns the temporal evolution of the genetic program within individuals and species. At the hardware designer's level—evolution of the physical circuit connection is usually implemented by *genetic algorithm*. Bit strings encode the circuit connections and evolution looks for the best connection for each task. The research field is called *evolvable hardware*. Ontogeny concerns the developmental process of a single *multicellular organism*—it is successive division of the mother cell, the zygotype, in which each newly formed cell possessing a copy of the original *genome* (*cellular division*), followed by a specialization of the daughter cells in accordance with their environment (*cellular differentiation*). *Genes*, the basic constituents of the genome, act on two quite different levels: they participate in the *embryonic process*, influencing the development of the phenotype in a given generation, and they participate in genetics, having themselves copied down the generations (*reproduction*).

The basic model of the developmental process is the *cellular automaton* [7]. For our purposes, two-dimensional cellular automaton, as the natural model of hardware implemented cellular system, is (non-formally) defined in this way: Cells (m x n of computing elements) are placed in the regular grid (see Fig. 1). Each cell is connected to the four neighboring cells called north, east, south and west neighbor. The automaton works synchronously—a new state of the cell is calculated from its previous state and the previous states of the cell's neighbors in each clock. Function of the cells, the initial states, and the boundary conditions depend on given task. *Cellular system* then, furthermore, involves description of (re)configuration mechanism, fault tolerance implementation, and other mechanisms which are related to the hardware implementation.

Cellular computing machines are quite different to the traditional parallel architectures: Instead of a powerful processors, scheduling, message passing, synchronization etc. in traditional parallel systems, they are inherently parallel, based on the local interaction of the millions simply working cells. The *cellular computing paradigm* (in Sipper's [6] term) promises to compute more efficiently—in terms of speed, cost, power, and solution quality for some special tasks as the image processing, NP-complete problem fast solution or physical phenomena simulation. Cellular computing shows three fundamental principles [6]: *simplicity*, *vast parallelism*, and *locality*. The goal is the regular structure of the identical cells, with easy implementation on the chip.

- *Simplicity.* The basic element of cellular computing—the cell—is simple. All cells are identical from the hardware implementation point of view. They differ in the stored configuration information, which determines

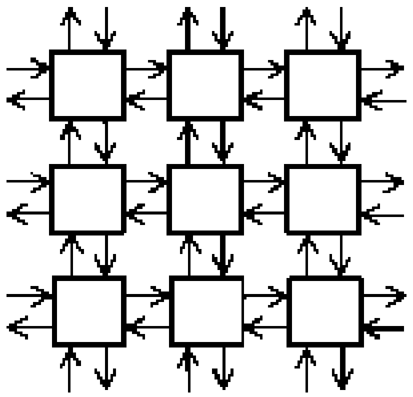

Figure 1. Two-dimensional cellular structure

behavior of cells. This behavior can be defined by lookup table, as the function of the cell's neighbors, by simple program, etc.

- *Vast parallelism.* In the traditional parallel computing domain, the term *massively parallel* usually describes machines consisting of several hundreds of processors and powerful communication mechanisms. Cellular systems suppose millions (and more) of cells.
- *Locality.* A cell communicates only with its neighbors. In our examples, regular grid with four neighbors per cell is used. Connection lines carry only small amount of information.

After this brief introduction to the cellular systems and cellular computing, the most important points of cellular system design are discussed. Special sections are dedicated to the (re)configuration mechanisms and fault tolerance implementation—from classical engineering, bio-inspired, and Macias's points of view. Advantages and disadvantages of these approaches are summarized and some conclusions are done.

2. Cellular system design

It is not possible to do any experiments with the real physical chips for cellular computing (meaning millions of cells on the chip), because these chips are not available nowadays. Only experimental systems with several cells exist [2, 1]. But simulators make a lot of work. Similarly to the quite different tasks, simulations can help to choose the best architecture and decide when the cellular computing is the best approach for the solution of a given problem. In cellular computing, simulations can be used to solve e.g. these design problems: the cellular function, structure, programming; the number of used cells; cellular connection, organization, interactions; configuration and reconfiguration mechanisms; fault tolerance implementation; input and output connection. It is important to note that it is necessary to develop the theoretical model and hardware implementation together in cellular computing [6]. These areas are closely related in this approach.

2.1. Programming

For most cellular computing models, it is possible to prove *computing universality* by implementing some serial universal machine in considered model [7]. But programming in this way leads to the total degradation of cellular machine parallel power. Classical programming approach based on *"divide et impera"* usually fails. Programmer has to solve the global task only by definition of local cell behaviors. And it is difficult. Similarly to evolvable hardware, cellular functions (e.g. lookup tables – rules) can be encoded into the bit strings and genetic algorithm then tries to find the best cellular functions [7] (machine genotype). Genes (several genes represent function of the cell) work at phylogenetic level—the best function of the cell is looked for. Genes also work on the ontogenetic level—they determine the developmental process of cellular system. Such constituted systems are bio-inspired in two levels of organization [6].

3. Reconfiguration and fault tolerance implementation

Cellular system configuration, reconfiguration and fault tolerance implementation are closely related problems and it is useful to describe them together. Three approaches—classical engineering, bio-inspired, and Macias's are considered and a special subsections are dedicated for them. The first solution is described in detail of hardware implementation, other principles are described at higher level. Next paragraphs comment some general concepts. Principles of locality and simplicity of cellular computing paradigm determine ways of configuration. Mostly the serial configuration of the configuration registers is permitted. The configuration information flow depends on the given approach.

Fault tolerance implementation is usually based on some kind of *redundancy*. In *time redundancy*, the task performed by faulty cell is distributed among its neighbors. Cellular systems have not any global control mechanisms thus they can not implement task redistribution and, furthermore, cells are too simple to do anything more. In this traditional way, time redundancy can not be used. *Hardware redundancy* uses *spare cells* to replace faulty ones. Two concepts are usually used: In *column* (or equivalently *row*) *elimination*,

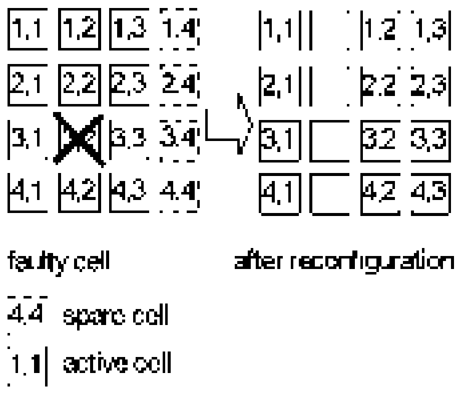

Figure 2. Spare column is used when a cell fails

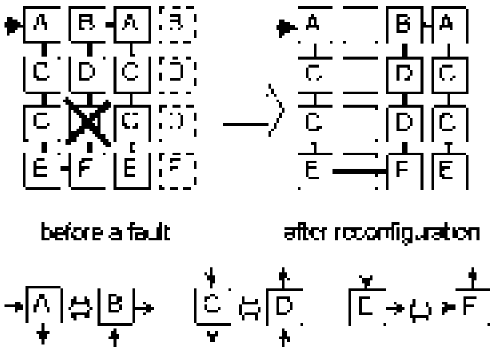

Figure 3. Principle of the configuration flow routing, where cells are configured serially. The producer switches one of the routing configurations A-F, which is inverted when a cell fails.

the failing of one cell provokes the elimination of the corresponding column, and cells are shifted to the right (see Fig. 2). This strategy eliminates many good cells when a fault occurs, but hardware implementation is simple. In *cell elimination*, only the faulty cell is eliminated. Such strategy provides a very efficient use of spare cells, but the complexity of interconnected circuits increases and regular topology is broken. It is in contradiction with the basic principles of cellular computing—simplicity and locality. Next text supposes that sufficient number of the spare columns is present.

Cellular systems bring a new view to fault tolerance implementation—they are *inherently fault tolerant*. When a cell fails, it still exhibits some behavior—e.g. all inputs and outputs are set up to the low logical level. When a configuration information is subject to evolution, a new solution (using faulty cell and other cells with modified function) can be found. Success depends on a given task. This effect can be considered as an implicit redundancy and it is the result of phylogenetic approach incorporation. It is quite different to other principles of fault tolerance, since they are inspired at ontogenetic level, where previous configuration is recovered. Generally, it is difficult to say, when the time or hardware redundancy is in principle used.

3.1. Tasks from the fault tolerance point of view

Two different kinds of tasks may be solved by two-dimensional cellular system. In the first case, cellular system is considered as a black box and only outputs are important for given inputs. It is usually difficult to decide how many cells are essential for a solution. In the case of evolvable machine, an explicit fault tolerance mechanism needn't be implemented, because the system is inherently fault tolerant. On the other hand, every cell (its state) is important for the system function—e.g. a pixel is mapped into a cell in tasks of image processing. Then when a cell fails, explicit fault tolerance mechanism must be activated.

3.2. Failure detection

Each cell contains internal diagnostic mechanisms, which activate an error signal when some defect occurs. It ensures distributed diagnostic without central control. In modelling, faulty cell is marked manually. In case of real implementation, e.g. module redundancy, error detection codes, and another logic should be considered. Details are not subject of the paper.

3.3. Engineering approach

These ways of the fault tolerance implementation are based on the column (i.e. hardware) redundancy. When a cell fails, every cell becomes a short circuits in the given column, some ERR signal value of this column and the global $CONF$ signal are set up, and a new configuration is downloaded from memory. The first spare column is used to replace the missing one.

In the *serial configuration*, configuration registers are connected in serial and the configuration bit stream is shifted through entire cellular array. Fig. 3 describes an initial and an after-repairing routing configuration. Figures 4 and 5 show hardware implementation: combinational circuit CC, which works according to the Fig. 3, selects configuration input. An internal function of the cell is based on appropriate configuration register bit selection. Identical cells

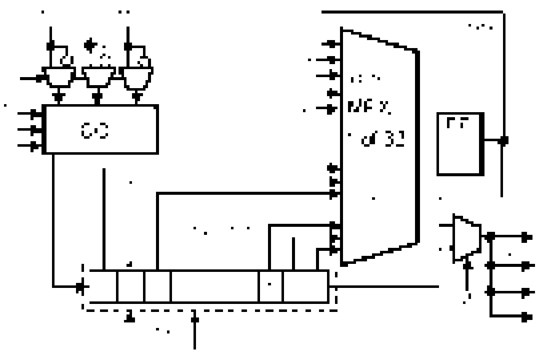

Figure 4. Internal logic of the serially configured cell

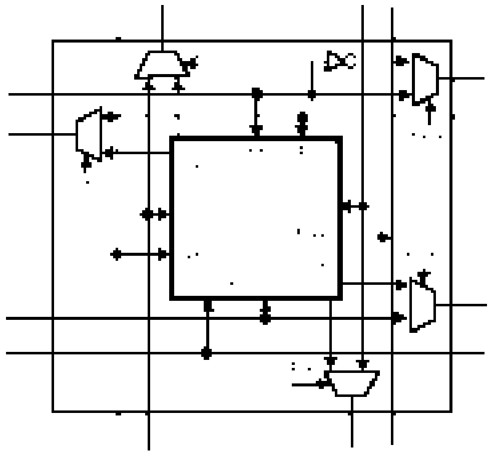

Figure 5. Cell's connection to neighbors

differ in the configuration information and the values of the three bit routing register ($M0$, $M1$ and $M2$). The routing values (switched by producer) and the $CHRC$ signal (CHhange Routing Configuration) determine configuration input of each cell. Configuration information is sent to every output of the cell. An invertor of the $CHRC$ signal in each cell ensures correct routing during configuration. Time of reconfiguration is proportional to the *C.m.n* product, where C is the number of bits in the configuration register.

In the *serially parallel configuration*, cells in the row are configured in serial, but rows concurrently. It is necessary to design a special configuration memory organization to ensure concurrent row configuration. Note that thousands of the rows can constitute a cellular system. The cell is simpler than in the previous example: the $CHRC$ signal, routing memory and the combinational circuit CC are omitted. The configuration bits flow from the west to the east. Time of reconfiguration is proportional to the *C.n* product.

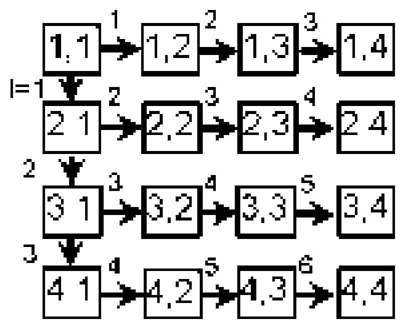

Figure 6. Configuration during cellular division process

3.4. Fault tolerance and embryology

The basic inspiration lies in the *embryonic development* of *multicellular organism* in nature. Two main bio-inspired mechanisms are used: *cellular division* and *cellular differentiation*. During cellular division a mother cell gives rise to at most two daughter cells, each obtains a complete copy of mother's genetic information [2, 4]. Cellular function depends on a part of this genetic information. Appropriate genes, which define configuration register, are selected according to the cell's position. Cellular differentiation is thus implemented using coordinate registers X and Y, which values are calculated from coordinates of the nearest west and north neighbors.

Fault tolerance (as in the previous approaches) is based on hardware redundancy—spare columns. When a cell fails, the appropriate column is eliminated. A new connection is established by recalculation of coordinates of the cells (to the right of the faulty ones only). New coordinates then determine new configuration register values. Configuration bits are not carried through cellular array, because cells have entire genetic program (of course, spare columns have to be configured). Thus a distributed reconfiguration mechanism is obtained.

This cellular system can be initially configured in serial, serial-parallel or during cellular division process [2]. Cellular division proceeds in discrete time steps, beginning with a mother cell placed at coordinates X=1, Y=1 (see Fig. 6). At the next time, this genome is copied to the two neighboring cells to the east and to the south. Process continues until the space is programmed. Coordinates are calculated during this process or they can be a part of the configuration bit stream. In the case of reconfiguration, no genes are transmitted. Faulty cells only signalize, that the right neighbors should recalculate their coordinates.

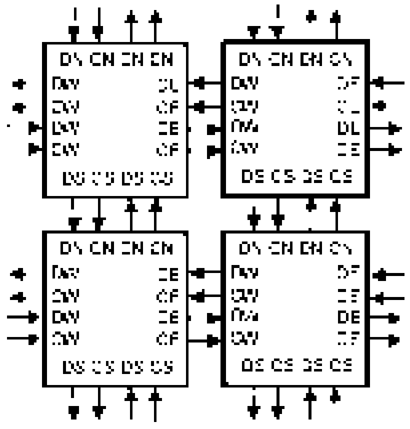

Figure 7. The Processing Integrated Grid

When cellular system contains millions of cells, it is not acceptable the cell holds the entire genetic information. On the other hand, when a system exhibits properties of *quasi-uniform cellular automaton* [7] (i.e. the most of cells have the same function) it is the best solution. Another solution leads to the restriction of genetic information of a cell [4]. The cell then may contain only its configuration bits and configuration bits of several neighbors (cell, containing genes of the cells of the same row, needs only one coordinate). But when several neighbor columns fail, it needn't be (generally) possible to reconfigure neighboring cells since the required genes are lost. Then a process of initial configuration must be restarted. Thus the entire array is configured in case of a major fault, while self repair allows a partial reconstruction in case of a minor fault. This approach exhibits fast and flexible reconfiguration, but additional hardware (against to engineering approach) is needed: configuration registers of neighboring cells, coordinate registers, logic for selection of the configuration and coordinate calculation.

3.5. Macias's cell

A quite different approach to the cellular system (re)configuration exhibits *The Processing Integrated Grid* (US Patent #5,886,537) or *PIG*, which is a massively parallel, fine grained, self-reconfigurable infinitely scalable system [1].

Figure 7 shows two-dimensional grid of identical cells, each with four neighbors. A cell has two inputs (data Din and configuration Cin) and two outputs ($Dout$ and $Cout$) from/to each of its neighboring cells. At any given moment, a cell is either in one of two modes: *data mode* or *configuration mode*. If all four of the Cin's are zero, then the cell is in data mode. The four Din bits are used as an address in a 16x8bit truth table. The 8 bits are output as data to the

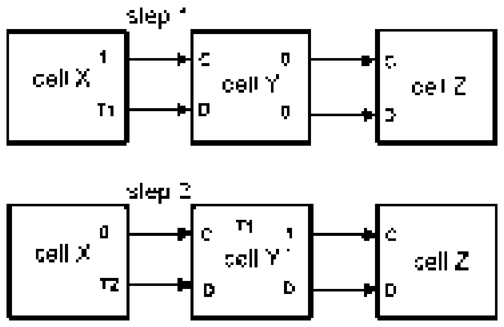

Figure 8. Cell X configures cell Z by first configuring cell Y to act as a router (with table T1) and then passing table T2 into Z via Y

four $Cout$'s and the four $Dout$'s. If one of the Cin's is a 1, then the cell switches to configuration mode and the D inputs are serially shifted into the cell's internal truth table. This allows one cell to write another cell's truth table, which subsequently affects that cell's behavior when it returns to D mode. As the new truth table is shifted into the cell, the cell's prior truth table is shifted out on its D outputs, and is available for reading. When a cell is in C mode, only the D inputs and outputs on sides where Cin = 1 are relevant [1]. Hence any cell can control the mode of any neighboring cell. By placing a neighboring cell in C mode and reading and writing that neighbor's D lines, a cell can read and write the truth table of any neighboring cell, and thereby configure it to subsequently perform any combinatorial function desired (after returning the neighbor to D mode). Since the neighbor's new combinatorial function can produce any desired C and D outputs, that neighbor can be configured to itself configure any of its neighboring cells [1]. Cell X (as seen in Fig. 8) configures non-adjancent cell Z by first configuring cell Y to act as router of X's configuration information (step 1), and then passing Z's desired truth table ($T2$) into Z via Y (step 2). Cell Y is first an object of configuration, and then becomes a configuration controller itself. This is called *code/data duality* or *self-configuration*. Because of the PIG's distributed configuration control, it can be used to study not only parallel execution of algorithms in hardware, but *parallel reconfiguration of hardware*. Moreover, the PIG can implement circuits, which create new circuits, which themselves create and modify other circuits. This simple cellular structure is extremely powerful and exhibits the inherent fault tolerance.

4. Conclusion

Many applications based on cellular automata are nowadays described and published, but only software simulations are usually used. The main benefit from the use of cellular approach is (except powerful parallel model, of course) the easy hardware implementation contrary to other computation models. Principles of simplicity, locality, and vast parallelism determine the paradigm of cellular computing and thus influence hardware design. During engineering design process, contrary to program simulators, more problems must be solved. This paper shows close relations between configuration, reconfiguration, and fault tolerance implementation of the vast parallel cellular system. Decision, what (re)configuration and what fault tolerant mechanism should be used, depends on the application. There are key questions: Should be used full (slow, using external memory, but with simple cells), partial (quick, using genes of the neighboring cell, but with complex cells) or implicit (using evolution or Macias's self-configuration) chip reconfiguration when a fail occurs? How many spare cells should be used? Should serial or parallel configuration be used? Is it useful to prefer principles from biology or traditional engineering approach?

Modelling and simulations can indicate many system features, but the final proof of the successful solution lies in hardware implementation of the chips with millions of cells. These chips and incorporation of phylogeny will probably open up inherent fault tolerance approach even more. Developed simulation models (consisting of several cells only) have been used for studying of described reconfiguration and fault tolerance principles.

This research was performed with the research intent CEZ: J22/98:262200012 "Research of information and control systems" and the Grant Agency of the Czech republic under No. 102/98/1463.

References

[1] N. Macias. The PIG paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. In *Proc. of The First NASA/DoD Workshop on Evolvable Hardware (EH'99)*, Pasadena, California, USA, 1999. IEEE Computer Society.

[2] D. Mange, M. Sipper, and P. Marchal. Embryonic electronics. *BioSystems*, 51(3):145–152, September 1999.

[3] P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, and G. Tempesti. Embryonics: The birth of synthetic life. In *Towards Evolvable Hardware. The Evolutionary Engineering Approach*, pages 166–196, Berlin, 1996. Springer-Verlag.

[4] C. Ortega-Sanchez and A. Tyrrell. MUXTREE revisited: Embryonics as a reconfiguration strategy in fault-tolerant processor arrays. In *Proc. of The Second International Conference on Evolvable Systems: From Biology to Hardware (ICES'98)*, pages 206–217, Berlin, 1998. Springer-Verlag.

[5] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Perez, and A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In *Proc. of The First International Conference on Evolvable systems: From Biology to Hardware (ICES96)*, pages 35–54, Berlin, 1997. Springer-Verlag.

[6] M. Sipper. The emergence of cellular computing. *IEEE Computer*, 32(7):18–26, July 1999.

[7] M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Berlin, 1997.