SABRE: A bio-inspired fault tolerant electronic architecture

P Bremner [1], Y Liu [2], M Samie [1], G Dragffy [1], A G Pipe [1], G Tempesti [2], J Timmis [2], A M Tyrrell [2]

[1] Bristol Robotics Laboratory, University of the West of England, Coldharbour Lane, Bristol. BS16 1QY.

[2] Intelligent Systems Research Group, University of York. Heslington, York, UK. YO10 5DD

E-mail: `paul.bremner@brl.ac.uk`

Abstract. As electronic devices become increasingly complex, ensuring their reliable, fault free, operation is becoming correspondingly more challenging. It can be observed that, in spite of their complexity, biological systems are highly reliable and fault tolerant. Hence, we are motivated to take inspiration for biological systems in the design of electronic ones. In SABRE (Self-healing Cellular Architectures for Biologically-Inspired Highly Reliable Electronic Systems) we have designed a bio-inspired fault tolerant hierarchical architecture for this purpose. As in biology, the foundation for the whole system is cellular in nature, with each cell able to detect faults in its operation and trigger intra-cellular or extra-cellular repair as required. At the next level in the hierarchy, arrays of cells are configured and controlled as function units (FUs) in a transport triggered architecture (TTA), which is able to perform partial-dynamic reconfiguration (PDR) to rectify problems that cannot be solved at the cellular level. Each TTA is, in turn, part of a larger multi-processor system which employs coarser grain reconfiguration to tolerate faults that cause a processor to fail. In this article we describe the details of operation of each layer of the SABRE hierarchy, and how these layers interact to provide a high systemic level of fault-tolerance.

1. Introduction

The demand for increasingly more 'intelligent' equipment has to be viewed through the explosion of their complexity. An important knock-on effect of increased complexity, however, is degradation in reliability: more complex systems are more prone to failure. For the vast majority of electronic devices, long-term fault free operation is an important requirement. Moreover, there are many systems for which correct operation is imperative, e.g. anti-lock braking systems, medical systems, space exploration, industrial control and shutdown systems. Hence, systems must be designed that are able to detect erroneous states and handle them such that they can operate correctly in the presence of errors and be fault tolerant.

The primary source of faults that effect digital systems is ionizing radiation. The two most common faults caused by radiation are single-event upsets (SEU), and single-event transients (SET), accounting for around 90% of faults [1]. SEU change the data stored in registers, while SET cause spurious data to appear within combinational logic affecting data calculations. Although both types of fault are temporary they have serious consequences for unprotected digital systems. There are also permanent single-event effects (SEE), but these only acount for some 10% of errors.

The ability of an electronic system to either carry on operating as normal in the presence of malfunction, or to 'roll-back' to a simpler safe mode is a well-established methodology, which focuses on replication of resources at the system or sub-system level. In nature, a similar situation does pertain; we have two lungs, two kidneys, two arms, and so on. However, nature achieves a great deal of its functional integrity by solving the reliability problem at the level of its primitive constituents: the cells. Multi-cellular systems develop from a single cell through the processes of cellular division and cellular differentiation. Differentiation is manifested by instantiating different functionality in each cell even though, at a fundamental level, any cell is capable of performing any of the functions (e.g. stem cells). Although local failures are common, and the body is under constant attack from harmful pathogens and environmental conditions, biological organisms protected by their immune system are highly reliable. The quite spectacular abilities that nature often displays, in this case via the immune system, drive us to attempt to replicate some of its features. If we can design systems that, like their natural counterparts, can withstand the occurrence of faults, and still function with little or no degradation in their performance, then it could also open up avenues for the application of electronic systems in more hostile environments than hitherto.

Prior work targeting this goal, Embryonics [2][3][4], POEtic [5] and related works [6][7], has adopted certain features of cellular behavior and organisation, transposing them to the two-dimensional world of integrated circuits on silicon. Fundamental artificial cells and systems with 'universal' cellular architectures have been created. These artificial systems have, like their biological counterparts:

(i) Cellular organisation, where an artificial system is formed by a homogeneous, initially identical, 'sea of cells', each containing the description of the system (the genome or DNA).

(ii) Cell division that 'creates' the number of cells required to define an organism (system) [7].

(iii) Cell differentiation that configures the functionality of each cell so that together they will provide the overall characteristic requirement of the organism (system) [7][3].

(iv) Error detection and correction at the intracellular level. Any mutation or fault in parts of the cell must be repaired to avoid erroneous system operation.

(v) Error detection and correction at the extra-cellular level. If repair at the intracellular level is not possible, then the cell must be replaced by a fault-free cell.

The work presented here builds upon the solid foundation of this prior work, varying from, and extending it, in several fundamental ways. In Embryonics, despite the interesting bio-inspired features, there are a number of engineering problems that have been under-estimated. Among them, cost-efficiency is the first and foremost issue: enormous electronic resources are required to achieve a maximally bio-plausible architecture, which prohibits Embryonics from being utilised for real world applications. This paradigm is overthrown in the SABRE project, where an artificial organism is created rather than an electronic copy of biology. Although (in a similar manner to Embryonics) the SABRE project utilises a hierarchical structure, with fault tolerance being initiated at the base substrate level, hierarchical divisions and the mechanisms employed are driven by the electronic system requirements. Thus, while biological inspiration is used for various mechanisms within the SABRE project, each layer is treated as a separate entity for the purposes of where the biological inspiration is drawn.

In the next section the features of each layer, and their various biological inspirations, are introduced. This is followed by details of the operation of the mechanisms that operate at the base substrate layer of SABRE, the Unitronics cells (section 3) and SMove, the transport-triggered architecture we have implemented on the second layer (section 4). Next, we look at the practical implications from instantiating a test application on an SMove processor (section 5). We then describe the top, multi-processor layer of SABRE (section 6). We then consider the issue of faults in the protection mechanisms, and how the hierarchical structure of SABRE might be leveraged to protect these systems (section 7). Finally we present conclusions and suggestions for future work (section 8).

2. Fault Tolerance Hierarchy and Biological Inspiration

SABRE has a 3 layered fault tolerance hierarchy: cellular layer, organ layer and organism layer, as shown in figure 1. This layered structure is inspired by the self-repair hierarchy in biological systems [8]. Cells are always under attacks from pathogenic beings and processes, and contain mechanisms to perform intracellular repair. However, if an infected cell cannot repair itself, it will perform apoptosis and release certain biochemicals to the surrounding tissue environment. When the concentration of these biochemicals is strong enough, it will trigger extra-cellular repair to replace the damaged cells in that tissue. Further, the immune system will also sense the abnormal local chemical level and help recruit stem cells to this damaged site.

In SABRE, each layer is a composed of a number of building blocks, each comprised of a set of components from the layer beneath. Each layer identifies, and deals with, faults at its level of granularity: when a building block is no longer able to deal with the degree of faults occurring within it, a message is sent to the layer it is a part of, the higher layer then tries to resolve the situation. It is important to note that the layer divisions

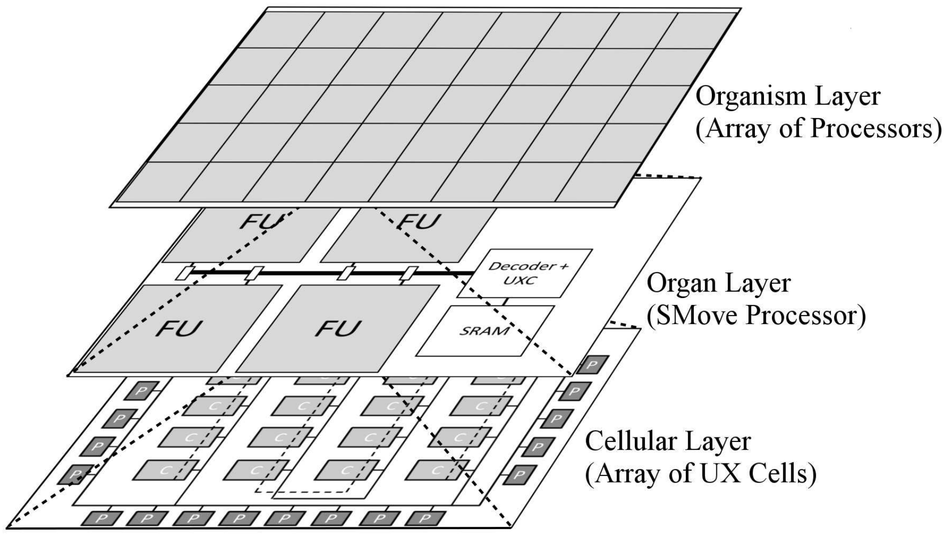

Figure 1: Hirarchy of the 3 layers of SABRE

are determined by the structure of the electronics, and thus not directly bio-inspired, different biological inspiration is however applied to the layers independently.

At the cellular level, we have used prokaryote cells as our biological inspiration, as opposed to the eukaryote cells used in Embryonics and related work [2][3][4][5][6][7]. The novel prokaryotic model which we have applied, uses the concepts of gene-compression and horizontal gene-transfer (HGT) found in colonies of prokaryotes [9] to provide a high degree of fault-tolerance, with much less redundant genetic information in each cell (than in Embryonics). In Embryonics the complete genome for the individual is stored in each cell, though only a small amount is expressed (by each cell); with any non-trivial circuit the amount of memory in each cell is prohibitively large. Conversely, our gene-compression method means that the redundant genetic information is the same size as the expressed genes, while still providing a high level of redundancy.

At the organ level, rather than a fully homogeneous sea of cells, we have utilised a transport-triggered architecture (TTA) we term SMove (SABRE Move), whereby the cells are pre-segmented into islands, managed by a controller, and hence facilitating efficient partial-dynamic reconfiguration (PDR). The organism level of the system, is a multi-processor array, in this layer each TTA processor is the building block, and the self-repair mechanism used is inspired by the development process and the innate immune system in vertebrates; it is important to note that because of the chosen bio-inspiration, SMove processors are also considered analogous to cells, i.e., when something is referred to as a cell it is important to note the context of the reference.

2.1. Unitronics Cells

Prokaryotes, such as bacteria, are single-celled microorganisms that are relatively simple in structure. A prokaryotic cell only has a single DNA molecule, which is a circular, super coiled, double-helix structure that has no ends. When only one of the two strands of a double helix has a defect, the other strand can be used as a template to guide the correction of the damaged strand, providing intracellular repair.

Although prokaryotes are single cell organisms, they have the ability to form cooperating communities such as colonies and biofilms, and hence perform tasks impossible for a single organism [9]. Of particular interest to the work presented here, is that biofilms have improved resistance to biocides when compared with single cells [9]. One cause for this is a form of acquired immune system by means of horizontal gene transfer (HGT); genes endowing a cell with resistances to biocides can be copied and transferred to other cells in the biofilm [10]. Additionally, extra-cellular repair of cells in a biofilm becomes possible by means of healthy cell duplication or recruitment of new cells.

Unitronics cells (**uni**cellular elec**tronics**), the primitive elements from which our system is comprised, utilise mechanisms which are inspired by the aforementioned biological characteristics, structures and behaviours of prokaryotes. The redundant genetic information stored in Unitronics cells for intracellular repair acts somewhat analogously to the second strand of the DNA double-helix in natural cells, guiding repair of a damaged configuration register. Similarly, an array of Unitronics (UX) cells is somewhat analogous to a biofilm, containing multiple *species* of cells, which use a process inspired by HGT to aid repair of damaged cells. HGT also facilitates a form of extra-cellular repair of the colony. New cell genesis is clearly not possible with electronics, so we must use redundant blank cells, which, using our HGT inspired process, are used to replace irreparable cells. It can thus be considered that utilisation of spare cells is recruitment of a replacement cell of the correct species, which is then modified using artificial HGT.

2.2. SMove, Transport Triggered Architecture

The transport triggered architecture (TTA) was created in the 1980's [11] and usually contains an instruction decoder, an interconnection network and a number of functional units (FUs), as shown in figure. 2 [12]. Functional units and the decoder are connected through data buses and address buses, a single data/address bus pair is called a slot, and a processor can have multiple slots to allow parallel processing to occur. Functional units have a single, usually globally addressable, input/output interface called a port. The connection of a port to the slot(s) is called a socket, the number of ports in a socket is flexible.

A TTA only has one instruction, which is: *move(destination, source)*. Operations are not explicitly expressed within the instructions, but implicit from the addresses of destination ports. Once the decoder retrieves the addresses from the instruction, it will open the corresponding source and destination ports; hence, the structure of a decoder is generally very simple.

In SMove there is a single slot, and each FU consists of an array of UX cells, and is

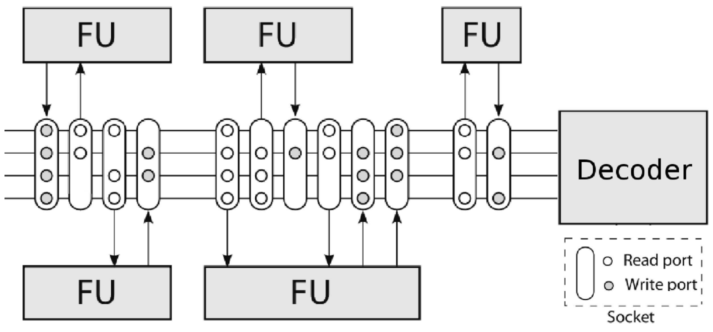

Figure 2: Schematic of a transport-triggered architecture

thus reconfigurable. Although not biologically inspired, the SMove forms an important part of the hierarchy, providing control of the UX arrays, and an additional biologically inspired fault-tolerance mechanism which we term the *reclaim algorithm*. The principle behind the *reclaim algorithm* is that a given fault will affect distinct configurations of a fabric differently; i.e., a fault that is detrimental in one configuration may be disregarded in a different one, due to the different utilisations of the available hardware. Hence, in the *reclaim algorithm*, different FU configurations are tested to see whether they might operate correctly on a damaged FU. The operation of this *reclaim algorithm* is somewhat inspired by, and analogous to, the degenerate nature of the immune system: degeneracy can be defined as "the ability of elements that are structurally different to perform the same function or yield the same output" [13]. Degenerate elements within systems have been observed to produce different contexts, thus making degenerate systems extremely adaptable to changes in their environment. Within the framework of the *reclaim algorithm*, the different contexts are the configurations of the fabric elements (cells), and the changes in the environment are the consequences of damage to the fabric.

2.3. Multi-Processor Array

In a multi-cellular organism, development is the process of cell division and differentiation, be it in embryonic cells, or stem cells in an adult individual. This intrinsic ability is encoded in the genome, and operates by means of a gene regulatory network (GRN), a network of chemical interactions that control gene expression [14]. One result of this lifelong developmental process is the immune system, embedded in the genes of which is the ability to not only remove harmful agents (i.e., a self-protection system), but also to influence the growth and repair of the organism by the inflammation response (i.e., a *self-maintenance* system) [15]. Hence, the two processes are intrinsically linked, with the developmental process being influenced by, and determining the behaviour of, the immune system. The main means by which this intrinsic link operates, is by cytokines (messenger proteins) produced by many immune cells, which act upon the system GRN [16]. The self-repair mechanism at the multi-processor array level is inspired by the biological development process, the immune system, and the mechanisms that operate within these systems.

We term our multi-processor array an artificial development substrate (ADS), whereby each processor is analogous to a cell in a multi-cellular organism, and can be used to instantiate either part of the system task or part of the artificial immune system (AIS), resulting in two subsystems. In the task subsystem, tasks are allocated to processors dynamically, and in such a way as to leave spare processors available, providing an intrinsic self-repair capability; when a processor is faulty, its role is dynamically assigned to a spare processor. The AIS subsystem detects errors and handles exceptional situations, providing additional protection inspired by the inflammation response in a biological immune system. Further, the two subsystems communicate using the same information exchange pathways that are analogous to the biological signaling materials, e.g. cytokines.

Previous approaches using similar inspiration [17][18] have focused on a much finer granularity, using configurable logic blocks as the cell analogues rather than micro-processors; doing so reduces the relative hardware overhead caused by the requirements of the developmental process. Further, their means of fault tolerance has relied on row and column elimination which is far less efficient than our immune inspired dynamic routing approach.

3. Unitronics Cell Array

In this section we describe the base substrate for our system, Unitronics, an array of digital electronic *cells* with mechanisms inspired by unicellular life. A Unitronics (UX) cell array is made up of two different types of cells; core cells comprising the main body of the array, which are surrounded by peripheral cells around the perimeter (figure 3). Core cells (section 3.1) are configured to implement specific data routing and processing functions, as defined by the genes stored in their configuration registers; additionally they are used to configure the internal data bus. Peripheral cells (section 3.2) manage the data flow to and from outside the array via a peripheral bus. The following sections contain a summary of the operation of a UX cell array, see [19][20][21] for more detail.

3.1. Core Cell Architecture

Core cells are the main component in the base layer of the SABRE fabric, they have 8 inputs and 4 outputs, and are made up of three elements:

(i) Memory: The memory consists of two independent banks of registers: configuration registers and repair registers. This is somewhat analogous to the double-helix structure of DNA, as the data in the configuration registers (the cells *genes*) determines the behaviour of the cell, and the data in the repair registers provides the required support in order to repair faults in the contents of the configuration registers. The contents of both sets of registers is verified by parity checkers, both during configuration and normal operation.

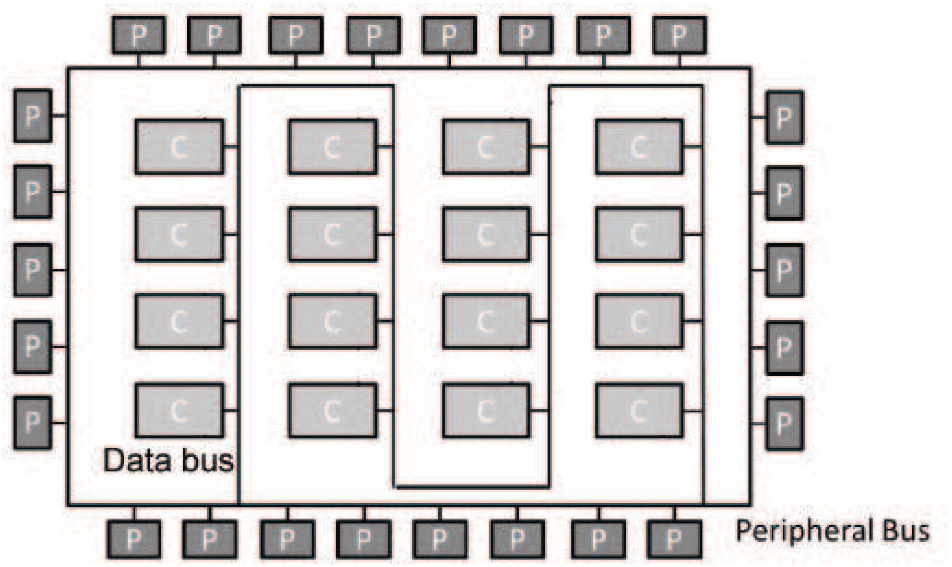

Figure 3: Schematic of a Unitronics cell array, see text for full description. Core cells are denoted C, and peripheral cells P.

(ii) Communication Unit: The Communication Unit controls data flow into and out of the cell, and allows configuration of the data bus that travels through it. Data can be routed into the cell not only from the data bus, but also from direct connections with neighbouring cells. This arrangement allows for highly flexible routing throughout the cell array.

(iii) Bit-Slices: The functionality of the cell is performed by two identical structures we term bit-slices (BS), which can operate independently or in concert depending on the task requirements. The available bit-slice operation modes are:

- Routing: A slice can be configured to route data from its inputs directly to its outputs. This can serve two purposes: firstly, data can be switched from one data bus-line to another for use by other cells; secondly, the data routing could be internal to the cell, as a routing slices outputs can be connected as inputs to the other slice.
- Function: The slice can undertake algebraic or logical functionality. In algebraic mode each slice can be a full or half adder. In the case when both slices are being used as full adders, the carry output of one slice can feed the carry input of the other slice. In the logical mode, the slice can realise any arbitrary three variable Boolean function. Additionally, the outputs of both slices can be combined to operate as a four variable Boolean function; this slice output multiplexing can also be performed between two cells to aid routing, and allow the composing of more complex logic functions.
- Data: Some of the configuration registers can be used to store input data. Reading and writing of this data can be controlled by another cell or one of the bit slices depending on the amount of data to be stored.

3.2. Peripheral Cells

Peripheral cells are much simpler than core cells, their role is simply to provide a configurable connection to external I/O channels via a peripheral bus. They also fulfill a vital role within the built-in self test mechanism used in the UX array described in the next section.

3.3. Built-in Self test (BIST) Mechanism

The BIST used in Unitronics cells utilises the fact that the bit-slices within the cell are a paired structure, using each bit-slice to verify the result produced by its partner. Allowing for checking of SET and permanent SEE occurring in the functional logic of the cell. This is achieved by each cell having a normal operational mode (NOM) and a test operational mode (TOM), which alternate during every system clock cycle by means of a control signal; the configuration register is a shift register, and the control signal triggers data shifting. In NOM the inputs of both slices are processed and their results are stored in an output register. In TOM the genes in the configuration register are shifted, and control inputs to the peripheral cells are changed, such that slice 1 will receive the signals that slice 2 had in NOM, and vice versa. At the same time the behavioural and mode genes are shifted such that the two slices alternate the functional role they are performing. The outputs of both modes are compared to verify that the cell is operating correctly. This continuous checking mechanism ensures faulty outputs are never sent to the outputs of the cell array, and transitory faults will be detected. Further, the swapping procedure ensures that all of the functional components are tested, including the bus lines which are loaded with different information by the peripheral cells in the two operational modes.

Transient faults on a bus line will be detected as TOM/NOM mismatch by each cell receiving inputs from that line. To prevent this causing erroneous cell repairs, each cell's communication unit contains a set of flip-flops and comparators for each input line; a comparison mismatch causes the data to be refreshed and the cell to repeat its calculation on the next clock cycle. This detection hardware also allows a cell to verify that data which is unmodified by the cell has not been altered by a SET on the bus lines, hence full bus test coverage is provided by the cells.

This BIST mechanism has a number of advantages over more conventional approaches. The main advantage, is that there is a greatly reduced hardware overhead compared to the conventional modular redundancy approach, whereby each functional module is duplicated (or triplicated) and the outputs of both are compared [22]; in UX the only additional hardware requirement is that needed for the switching operation. However, this advantage comes at the expense of a reduction in operating rate of the hardware due to the alternating between NOM and TOM, i.e., time redundancy rather than hardware redundancy is employed. Traditional approaches to time redundancy use some means of encoding and decoding the inputs to generate data for a comparison that will detect faults [23][24]. They operate on the premise that the encoded data exercises the device in a different way to expose faults. However, they rely on the structure of the tested circuit being such that the encoding method exposes any faults, which (particularly when evolutionary computation is used for synthesis) is not guaranteed to be the case; our BIST method on the other hand, has no such requirement.

3.4. Self Repair Mechanism

When a cell is detected as being faulty a repair algorithm is executed by a controller external to the array (UXC) in conjunction with internal cell controllers (ICCs) present in each cell; to ensure fault free operation of the UXC and ICC they are both designed with a TMR structure, with self-testing voters. The repair algorithm that is used depends on whether a hard fault (i.e., damage to the hardware) or soft fault (i.e., flipping of a bit in the cell's genes) is detected. Soft faults are detected by means of parity checkers, and if such a fault is detected then the genes simply need to be reloaded into the cell; however, if a parity error persists after reloading, then the fault is deemed to be hard instead. The integrity of the parity checker result is ensured as the two halves of the genome (controlling the 2 slices) are separately parity checked, and the results compared in TOM and NOM.

The majority of hard faults are detected by means of the BIST mechanism previously described. The hard fault recovery algorithm proceeds as follows:

- All cells following the faulty cell are shifted along the bus, with the end one occupying a spare cell, and thus a spare cell directly after the faulty cell is created.
- Recover genes of faulty cell (detailed in the following paragraphs).
- Load genes into new cell; the bus width of the cell array is such that all genes can be loaded in parallel.
- Apoptosis of old cell by setting the bus lines to bypass it.

In either fault case, the genes (i.e, cell configuration vector, C_{CV}) of the faulty cell must be recovered, and this is done by means of *correlated redundancy.* We use the term *correlated redundancy* to describe the method used to store multiple redundant copies of each cell's genetic information throughout the UX cell array; in spite of the fact that the repair register in each cell is only the same size as the configuration register. In order to do so, the correlation between similar genotypes is used, effectively compressing the information. The first stage of this compression is to group the cells in an array based on the similarity of their genes, i.e., cells with similar genes should be grouped together, we term these groups *clusters*; clusters are purely for self-repair, and ignore the physical location of cells. Each cluster is distinguished by a *shared value* (C_{SV}) which the genes of cells within the cluster are within a hamming distance threshold of. Hence, each cell stores a *differential value* (Δg) in its repair register, which is its difference to the C_{SV}, and it is calculated by:

$$\Delta g = C_{SV} \oplus C_{CV} \tag{1}$$

This differential value is also used for another grouping, we term *colonies*, cells with the same Δg (irrespective of C_{SV}) are said to be in the same colony, (as with clusters) this is purely for self-repair, and ignores the physical location of cells. In order to recover the C_{CV} of a damaged cell, the C_{SV} of the cluster to which it belongs is calculated by one of the other cells in its cluster using 2, then its Δg is copied from another cell in its colony; hence, the C_{CV} can be calculated using 3, and loaded into the prepared spare cell. Each cell contains the required hardware to carry out 2 and 3 (largely consisting of a set of XOR gates).

$$C_{SV} = C_{CV} \oplus \Delta g \tag{2}$$

$$C_{CV} = C_{SV} \oplus \Delta g \tag{3}$$

It is a requirement to have multiple copies of each Δg across multiple clusters, with each cluster containing several individuals. This is done by specifying a hamming distance threshold of a shared value which an individual must be within for cluster membership, a lower threshold increases the chance of colony formation but reduces the chance of cluster formation; hence, the threshold must be carefully selected and is largely application dependent. A consequence of defining a threshold is that some cells within a design have a C_{CV} that is not sufficiently similar to any other cell to form part of a cluster. For such cells, *direct redundancy* is used. In this case two cells are paired, and each stores the others C_{CV} in its repair register, i.e., the C_{SV} is all zeros, and the Δg is the other cell's C_{CV}. However, the gene recovery sequence operates largely unchanged. One method we have investigated to aid cluster and colony formation is to divide the genome into subsections that can be compared separately, i.e., fewer bits means an increased chance to find commonalities (section 5.1.1).

Cells in a cluster can be considered to belong to the same *species*, varying from the base genotype (C_{SV}) by small genetic differences (Δg); in nature this occurs due to adaptations to the cell's environment (i.e., genetic learning), whereas in an artificial system the differences are inherent. In bacterial colonies, these genetic differences can be transferred between cells by means of horizontal gene transfer (HGT), modifying the genes of cells in which HGT occurs. In natural systems, this process is thought to be part of the reason behind the increased antibiotic resistance of biofilms [9]. Within a UX array, an analogue of HGT occurs by means of the intra-colony transfer of Δg as part of the repair process, ensuring that newly recruited cells are differentiated from their base *species* to perform the appropriate function.

In order to acquire the C_{CV} and Δg, from the cluster and colony respectively, tags are assigned to each cluster (T_{SV}) and each colony ($T_{\Delta g}$), and hence, by locating matching tags within the array the values can be acquired; T_{SV} is stored in the configuration register, and $T_{\Delta g}$ is stored in the repair register. In order to ensure that the tags (upon which the repair system depends) have not been damaged, they are monitored using parity checkers and, if they are error free, repair proceeds as described. In the event that an error is detected in the tags, a content addressable memory (external

Number of NAND gates in design 1	Overhead in design 2	Overhead in design 3	Overhead in design 4
3593	180.2%	78.1%	49.2%

Table 1: Synthesis for a single UX cell. Design 1 is only the functional components. Design 2 is the functional components, the BIST, and the UXC. Design 3 is design 2 without the flip-flops and comparators in the communication unit. Design 4 is design 3 without the UXC.

to the cells) is used, within which the tags are stored sequentially, thus, the faulty tags are recovered by using the values of those in the adjacent cells as a reference.

3.5. Hardware Overhead of Test and Repair Mechanisms

In order to investigate the hardware overhead associated with the proposed self-test and self-repair mechanisms, and their associated controllers, we have synthesised a Unitronics cell with and without those mechanisms using the Synopsys Design Compiler. The results shown in table 1 are using the UMC 90nm LL Low leakage library, with a clock constraint of 100MHz. In this library, the width and length of a NAND gate (ND2RX1) is $1.12\mu m * 2.8\mu m$, giving an area $3.136\mu m^2$. This area of the NAND cell is used to divide the total area of each design to give the number of equivalent NAND gates.

In table 1, design 1 is a Unitronics cell with only functional components. The other designs exclude certain components, and these are compared with design 1 so percentage hardware overhead can be calculated. To put these figures in context, triple modular redundancy (TMR, the most prevalent testing method currently used, where each system is triplicated and the results combined using majority voting hardware [22]) requires 200% overhead plus a non-trivial amount of extra for the voting circuitry; if voting is done at a functional (as opposed to system) level, overheads as high as 700% are reported [25]. Further, in Unitronics, only one external controller (UXC) is needed for several arrays of cells, so its overhead of 28.9% (found by comparing design 3 and 4) relative to one cell can be considered negligible when compared to several arrays of several hundred cells each. Hence, we suggest that the overheads for UX compare highly favourably with those for TMR. It can also be observed that the bus line testing in each cells communication block requires around two thirds of the overhead in each cell (design 2 - design 3 + design 4); this observation motivates us to seek a more efficient solution for safe guarding the buses in future work.

The spare cells required to allow replacement of faulty cells must also clearly be considered overhead. The percentage of overhead they incur is clearly dependent on how many spare cells need to be provided, i.e., the number of faults that can be recovered from. It should be noted that our cell replacement algorithm requires fewer spare cells for a given level of fault tolerance, than the significantly more costly row or column elimination previously applied in cellular systems [2].

3.6. Unitronics Arrays Within the Fault Tolerance Hierarchy

A UX array has a fixed number of cells, as they are encapsulated by a peripheral bus; hence, the number of spare cells within an array (after a desired functionality is configured) determines the number of faults that can be tolerated. The UXC, that operates the cell level fault repair algorithm, is able to detect whether any spare cells remain within a UX array; when no spare cells remain in the UX array any further faults cannot be repaired, thus endangering system integrity. Consequently, when no spare cells remain, a signal is sent to the next level up in the hierarchy, i.e., the SMove controller, in order that this situation can be handled.

While unlikely†, permanent faults in the bus lines cannot be recovered from. These unrecoverable faults are detected by persistant error detection in the comparitors in a cell's communication unit, i.e., repeated retransmission of data does not correct the error. This is handled by sending a signal to the SMove controller, triggering its fault handling procedure. We handle the faults in this way as replicating the communication bus with the associated multiple voters in each cell's communication unit would create a very large overhead, coupled with the low chance of the fault occuring. Details of the SMove, and the fault tolerance algorithms operating upon it, are described in the next section.

4. SMove Transport-Triggered Architecture

At the second level of fault tolerance hierarchy we have implemented a TTA processor, called SMove (SABRE Move). Each FU in SMove is comprised of an array of Unitronics cells as described previously. However, ports are implemented with dedicated hardware, i.e., not comprised of UX cells, hence an open port connects to the peripheral cells of a UX array. In order to test for SET in the bus lines in the slot, each message is appended with an error detection code and the ports contain parity checkers, parity mismatch triggers a repeat message request to be sent to the controller. In future work we will investigate the use of error correction codes and the relative hardware and time overheads associated with the two methods. Additionally, in SMove the simple decoder in a standard TTA is integrated into a device control unit that also includes a UXC (which monitors and controls all the cells in all the FUs in that processor), and self-diagnosis logic.

In the SMove, there are two working modes, the execution mode and the self-test mode, which are running cyclically. The entire program is sliced into a number of pieces and the length of each piece is decided by the empirical data of the mean time to failure (MTTF) of each FU. Running at the end of each piece, the SMove saves the current states and evokes a self-test routine. In this routine, the SMove polls the statuses of all

† SEU account for only 10% of faults, and of those, only electro-migration will affect the bus lines

FUs (whether there are any spare cells) and if there is no fault reported, it will restore the previously saved states and jump back to the normal execution mode. If a fault is reported, then the *reclaim algorithm* is triggered (see section 4.1), and its operations are divided up such that they may be performed when the operational mode is inactive; the system clock cycle is assumed to be faster than the required machine cycle, with a duty cycle of less than 100%.

4.1. Reclaim Algorithm

As the number of cells within an FU is finite, the ability to reuse areas where faults have been detected will allow fault tolerance with a considerably reduced amount of additional resources, and enable a larger number of faults to be tolerated, extending the lifespan of the system. The basis for doing so is that a given fault will affect distinct configurations of a fabric differently; i.e., a fault that is detrimental in one configuration may be disregarded in a different one, due to the different utilisations of the available hardware. This new procedure is termed the *reclaim algorithm* (summarised in figure 5). It is triggered by a function unit running out of spare cells (figure 5b), or a permanent fault being detected in the buses. When this occurs, an associated flip-flop is set to indicate the vulnerable status of the functional block to the SMove controller.

The reclaim algorithm then proceeds in three stages:

(i) Address reassignment
To begin the reclaim process the functionality of the damaged FU must be assigned to an available spare function unit (figure 5c) by remapping its address. The addresses of ports, which are called P_{ID} (port ID), are physically determined during fabrication, and the addresses used in the instructions, which are called V_{ID} (virtual ID), refer to a lookup table, where the content of each memory location in the table is the P_{ID}. By simply swapping the content in the address-mapping table, the data flow between FUs could be altered without re-writing the program (figure 4). The process operates in two stages: firstly, the configuration strings for the damaged block are used to configure the spare FU; secondly the addresses of the two FUs are reassigned, the address of the original function unit is assigned to the newly configured spare FU, and an address utilised in the program memory for the reclamation process is assigned to the damaged FU. This address reassignment process means that the reclaim algorithm can be performed online, i.e., correct function is ensured by the configured spare FU and configuration testing is performed concurrently (the duty cycle is assumed to be such that configuration testing instructions may be transmitted without affecting performance).

(ii) Configuration testing
The second stage of the reclaim algorithm is to sequentially test different configurations of the FU to find out whether they are able to operate in a fault free manner on the damaged FU (repetition of figure 5d for each configuration). Multiple distinct configurations are possible for a given FU functionality, and there

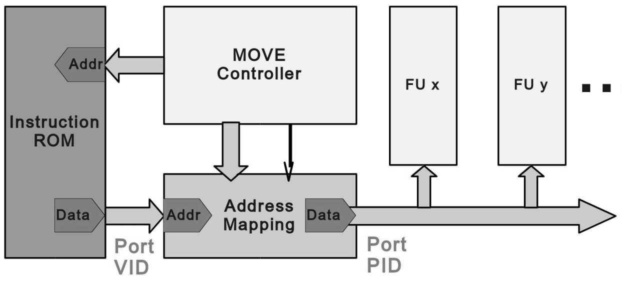

Figure 4: Schematic of the address-mapping mechanism used to dynamically assign ports to function units.

is likely to be more than one FU functionality in the system. For each available configuration, the appropriate genes are used to configure the FU, next a sequence of test vectors are sent, and faults are detected and responded to as previously described. When a fault is detected, the response signals the restarting of the test vector sequence following successful repair. Hence, if a configuration has no spare cells due to repair of detected faults, it signals to the controller in the normal manner. If a configuration is found that requires less cell excision than there are spare cells, the reclaim is found to have been successful and the algorithm progresses to the third stage. If all configurations have been tested with no success, reclaim is not possible and the algorithm is halted; in this case, the damaged FU is assigned a null address, and thus is permanently removed from the system.

(iii) Final address assignment
In the third stage, addresses are assigned to the FUs in order to return one FU to spare status (figure 5e), the procedure followed depends on the functionality of the successful reclaim solution. If the successful reclaim solution is of the same functionality as the faulty FU's original configuration, then the faulty FU is reassigned its original address, and the spare FU is returned to spare status. On the other hand, if the successful solution differs from the original functionality, then the current FU utilised for that functionality will become the spare block, and its address will be assigned to the reclaimed block; the previous spare FU will remain assigned the address of the previously damaged FU functionality.

4.2. Hardware Overhead of Test and Repair Mechanisms

The hardware in the SMove controller that can be considered as overhead is the controller, the parity checkers in each port, the configuration memory and the spare function units. The controller is of a similar complexity to the UXC so, while we have not synthesised it, it is clear that it represents a similar amount of hardware overhead, i.e., negligible, at less than a third of a UX cell in terms of area for control of likely hundreds of cells in the processor as a whole. Similarly, the parity checkers used in each port are significantly less complex than a UX cell so again they represent negligible overhead.

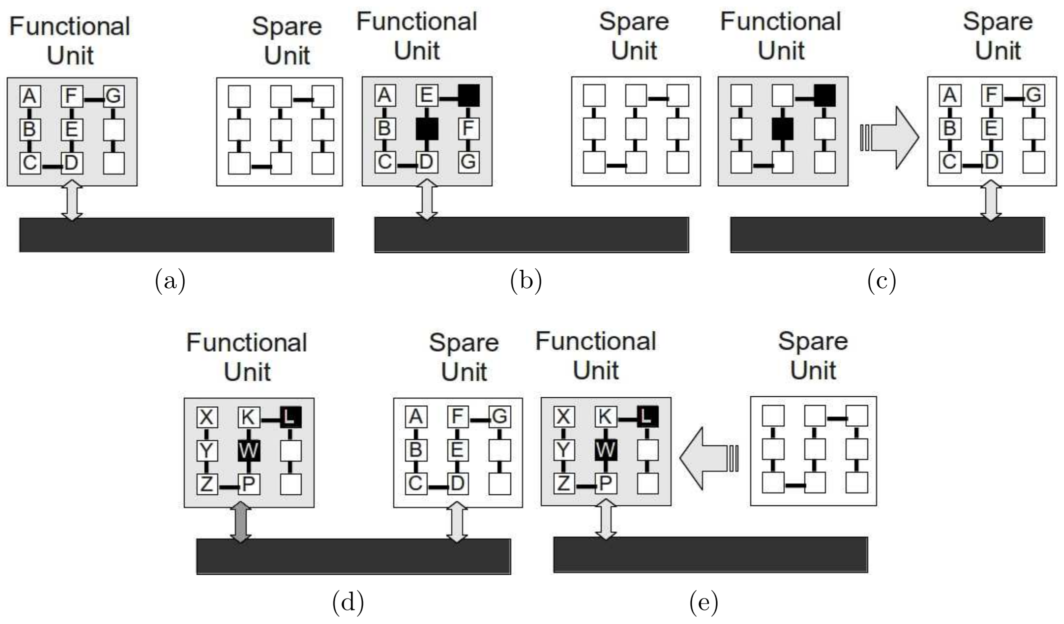

Figure 5: Reclaim Procedure. In the initial state (a) there are no faulty cells and 2 spare cells in the configured FU, and 1 spare FU available. 2 faults occur (black cells) causing use of the spare cells (b). Reclaim is triggered, and the functionality of the damaged FU is shifted to the spare FU (c). Alternate configurations are tested (d), repeated until a working configuration is found. Addresses are reassigned to free up the spare unit (e).

Hence, the main overhead of the system is the required additional SRAM to store the program instructions and configuration information for fault recovery, but this is highly application dependent so difficult to quantify. The reconfiguration information is equal in size to the configuration registers for all configured FU times the number of alternate configurations required for each. As an example consider that the (configuration and repair) registers in a single UX cell total less than 13 bytes, so a small 128Kb SRAM could contain configuration for around 10K cells; we envision a typical FU being around 100 cells, so that is around 100 FU configurations. The overhead incurred by the spare FUs is dependent on the size of an FU and the number of spare FUs that are to be made available, regardless this is still significantly less than using duplicates of the whole system, as in TMR, to allow repair of the system.

Another important feature of SMove is the reclaim algorithm, which increases the efficiency of the amount of overhead compared to the number of faults that can be tolerated before the whole system is unable to cope. This is particularly relevant as typically around 90% of faults are of a transitory nature [1] (either SEU or SET), so by reconfiguring an FU, cells killed as the result of transitory faults are likely to be returned to use. Additionally, in different configurations the bus lines are utilised differently, so permanent faults in the bus lines may be able to be masked.

4.3. SMove Within the Fault Tolerance Hierarchy

In an SMove processor there are a finite number of reconfigurable FUs, at least one of which must be left spare to enable PDR and the *reclaim algorithm*. Hence, an individual processor will be able to tolerate a finite quantity of faults. In its test cycle, the SMove controller is able to detect the number of spare FUs remaining, and if any spare cells within configured FUs remain. When there are no more spare resources available subsequent faults are not repairable and will cause the processor to fail. Consequently, a signal will be sent to the next level up in the hierarchy, i.e., supervising processors in a multi-processor array.

While unlikely, permanent faults in the bus lines cannot be recovered from. These unrecoverable faults are detected by repeated parity mismatchs in the error detection codes of messages from a port. These faults are also reported to supervising processors. Methods for implementing redundant slots, and using time redundancy in a multi-slot system to ensure fault recovery are the subject of future work. Details of the multi-processor controller, and the fault tolerance algorithms operating upon it, are described in section 6.

5. Example Application of a Single SMove Processor

We have instantiated a simple robot controller on an FPGA simulation (on a Xilinx Virtex-5 LX110T FPGA) of an SMove processor as a proof of principle for the first two levels of the hierarchy. The robot is an e-puck [26] operating as a simple Braitenberg vehicle [27] with distance sensors cross coupled to provide inhibitory signals to the drive wheels so that it produces an obstacle avoidance behaviour (figure 6); the sensor values must be thresholded, and multiplied by a gain to provide suitable inhibitory values, and these tasks are undertaken by SMove FUs. The FPGA based simulation of the SMove processor consists of FUs for both the thresholding and scaling of the sensor signals, one of which is made up of 15 UX cells and the others use the standard FPGA logic. In order to reduce the required FPGA resources, only 4-bit data is utilised; consequently, 4-bit sensor data will produce an 8-bit drive value, and the calculation is broken down into two FUs, one providing the 4 least significant bits of the answer, and the other the 4 most significant bits (most significant nibble, MSN). Hence, the FU that is utilised for this example (i.e., that runs on the simulated UX cells) provides the MSN of a simple scaling of the sensor data by a factor of ten. Communication between the FPGA and e-puck is over Bluetooth via a PC.

5.1. Implementation

In order to generate multiple novel configurations for each function unit, as required by the *reclaim algorithm*, we have utilised an evolutionary computation approach; by contrast, traditional design approaches will typically yield very few alternative solutions. The task is decomposed into functional and routing evolution. For functional evolution

Figure 6: The e-puck robot [26].

we have used a version of Cartesian Genetic Programming (CGP) modified in order to accommodate the Unitronics cell design [28][29]. The evolution of the data routing (for a functional solution) is performed using a simple generational GA. The final routed solution for the MSN uses 11 cells. Details of these processes are omitted here for brevity, but can be found in our earlier papers [28][29].

5.1.1. Cluster and Colony Formation In order for the repair mechanisms to function on the implemented program, the genes of the cells need to be correlated, and hence form clusters and colonies. A simple heuristic (Algorithm 1) has been designed in order to find clusters. As Algorithm 1 states, the gene-string with the lowest total hamming distance to all other members of GS (the gene-string array) is removed from GS and used to start the first cluster as the C_{SV}; all cells with a hamming distance (H) to the C_{SV} below a threshold (H_{thresh}) are also moved from GS into the cluster. In any given cell's configuration, some of the bits in its gene-string are not used and hence these are assigned *don't care* symbols, for the purpose of hamming distance, they are considered to match any symbol. However, once a cluster has been formed the *don't cares* must be resolved. This is done by examining the frequency of 1s and 0s at each bit location, and the most frequently occurring symbol is used to resolve the *don't cares*, if there is no majority a 0 is assigned. When this has been done, the Δg values are calculated for each member of the cluster. This process is repeated until all members of GS have been moved to a cluster, and had Δgs calculated. As previously stated (section 3.4), some gene-strings may be too different from the others to form part of a cluster and, in this case, direct redundancy is used; pairing of genes for this purpose is performed after all cluster assignment is completed. In implementing the MSN FU, it is immediately apparent that a very high threshold is required in order to form clusters and as a consequence, no colonies are found. This is a result of the low number of cells in the FU (11), and the large number of bits in the genome (46). In order to address

ALGORITHM 1: Frequency Number Computation

```
Input: Cell gene-string array GS in ascending order of total hamming distance
       to all other GS members, some bits in the gene-strings may be don't
       cares
Output: Cell gene-strings with resolved don't cares and assigned to clusters,
        $\Delta g$ for each gene-string
$index = 0$; $H_{thresh} = 5$;
repeat
    $\alpha$ = GS[0];
    Add $\alpha$ to $Cluster[index]$ and remove from GS;
    for Each string $\beta$ in GS do
        $H = \text{bitcount}(\alpha_{C_{CV}} \oplus \beta_{C_{CV}})$;
        if $(H < H_{thresh})$;
        then
            Add $\beta$ to $Cluster[index]$ and remove from GS
        end
    end
    for Each string $\beta$ in $Cluster[index]$ do
        Resolve don't cares;
        $\beta_{\Delta g} = \alpha_{C_{CV}} \oplus \beta_{C_{CV}}$;
    end
    $index$ ++;
until All gene-strings assigned to a cluster;
```

this issue, we bisected the genome for the purpose of generating C_{SV}s and Δgs, i.e., each cell effectively has two 23-bit gene-strings for the purpose of fault detection and recovery. Hence, using an empirically established H_{thresh} of 5, five clusters were formed, with three colonies. In order to improve colony formation, shared values were adjusted by hand to ensure repeated difference values; in the future we plan to develop a method for automating this post-hoc adjustment.

5.2. Fault Injection and Recovery

The length of time that it takes for the system to recover from a detected fault is critical to the utility of the SABRE fabric, i.e., faults must be repaired fast enough for system operation not to be detrimentally perturbed. The length of time a fault takes to be repaired depends on the nature and number of faults injected. The length of time in clock cycles for *soft* and *hard* faults can be calculated using 4, and 5 respectively. $F_{C_{CV}}$ is the number of faulty gene-strings, and $F_{\Delta g}$ is the number of faulty difference values; the additional term S in 5 is the number of shifts that must be done, and is equal to the number of faulty cells. For example, to repair a soft fault takes 7 clock cycles, and it takes 8 clock cycles for a hard fault. In the case where the data tags in a cell have become damaged, it takes an additional clock cycle for each tag that must be retrieved. On our test system the FPGA clock is running at 20MHz, so repair of a single hardware fault takes 0.4ms. The update frequency of the e-puck is (due in large part to communication delays) only 5Hz, hence the robot was observed to operate unperturbed, even when 4 simultaneous faults were injected (the maximum number of hard faults from which the system can recover in this example).

$$t_h rep = ((2 * F_{C_{CV}}) + 1) + ((2 * F_{\Delta g}) + 1) \tag{4}$$

$$t_s rep = (S + 1) + ((2 * F_{C_{CV}}) + 1) + ((2 * F_{\Delta g}) + 1) \tag{5}$$

5.3. Reclaim of the Damaged FU

The island we have instantiated on the FPGA simulation of a UX cell array has 15 cells, hence when 4 faults are injected the reclaim algorithm is triggered, i.e., the implemented function uses 11 cells, so recovery from faults in 4 separate cells means there are no more spare cells. During reclaim the functionality of the damaged FU is assigned to a copy implemented using the standard FPGA logic. For clarity we have only considered one of the faults injected to be permanent, and the other 3 are transitory, i.e., when alternative configurations are tested a fault only remains in one cell. A permanent stuck-at-1 fault is injected on the output of bit 4 in the right hand bit-slice (shown in red in figure 7b), these genes control routing for lines into the full adder performed by the other slice; hence, the fault causes incorrect data lines to be routed, and TOM outputs differ from NOM outputs. The faulty FU is successfully reconfigured using an alternative evolved configuration that uses 12 cells. In the cell where the fault occurs, in the new configuration, the left-hand slice is still a full-adder, but different lines must be routed, consequently the *behavioural genes* are 10010001; the different genes mean that the fault does not affect the operation of the cell. It is important to note that the *reclaim algorithm* allows cells subject to transitory faults to be reused by the system. Without its use they would be permanently excised, despite the temporary nature of the faults.

6. Multi-Processor Array

The top level of our fault tolerance hierarchy is a multi-processor array, upon which we have implemented a dynamic, self-organising, task allocation algorithm (section 6.1), and an artificial immune system (section 6.2). The dynamic task allocation mechanism is inspired by two biological networks: a protein diffusion network operating across the whole organism, and a gene regulatory network within each single cell [30]. Correspondingly, in the processor array, there is an inter-processor configuration status diffusion network (CSN) and an array of configuration regulatory networks (CRNs). The CSN requires configuration status messages (anologous to cytokines and other messanger

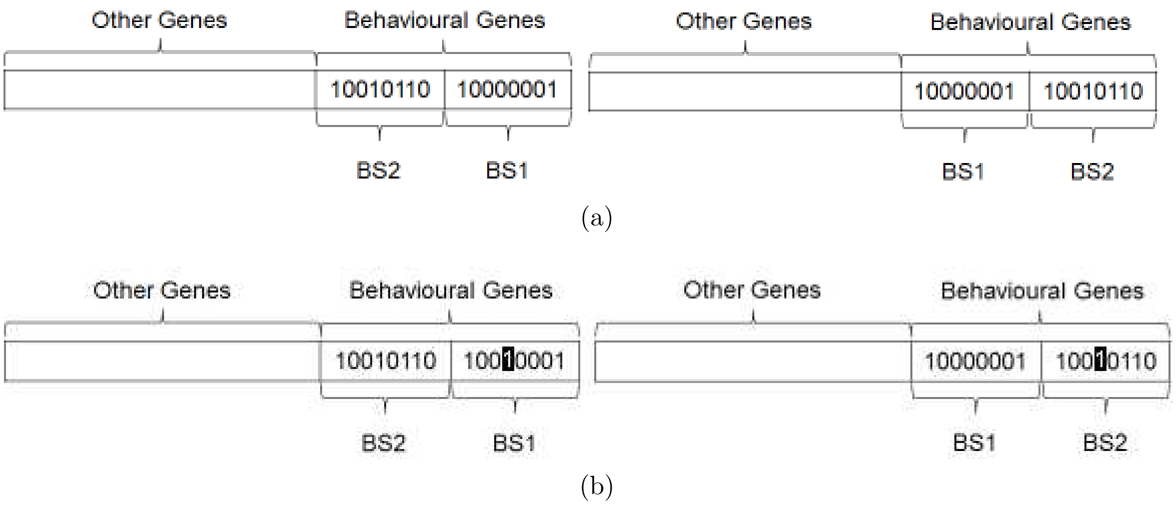

Figure 7: In BIST the behavioural genes controlling the two bit-slices are swapped (NOP on left, TOP on right) to test that the functionality performs identically on both slices. In (b) a stuck at 1 fault occurs on the output of gene 4 in the right-hand behavioural genes register (red), and hence the cell-outputs in TOP mode differ from NOP mode.

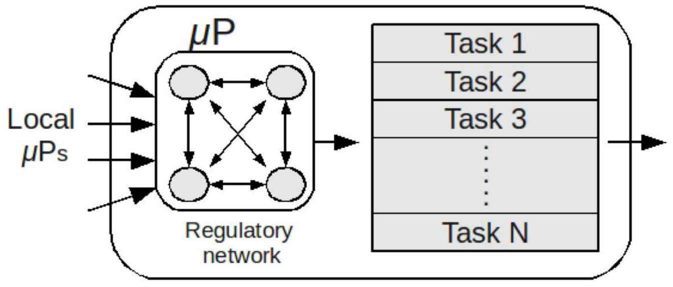

Figure 8: The task selection mechanism

proteins in the biological system) to be regularly broadcast, and be processed differently from workflow data, hence we have implemented two dedicated buses.

The CRN in each SMove processor, is a dedicated FU that processes incoming network messages in order to determine the role that it should undertake, and the messages it should transmit (figure 8). Each processor can undertake a workflow or immune task (i.e., an active proccessing-unit, AU), or to maintain a non-configured state (i.e., a dormant proccessing-unit, DU); in order to do so each processor must store all possible tasks.

6.1. Task Allocation

Task allocation on an unconfigured processor array, analogous to specialisation of stem cells in biological development, is initiated by configuration of some of the required AUs (typically the input and output processors). The configured AUs broadcast status related Task-ID messages (T_{ID}s) to neighbouring processors with an associated

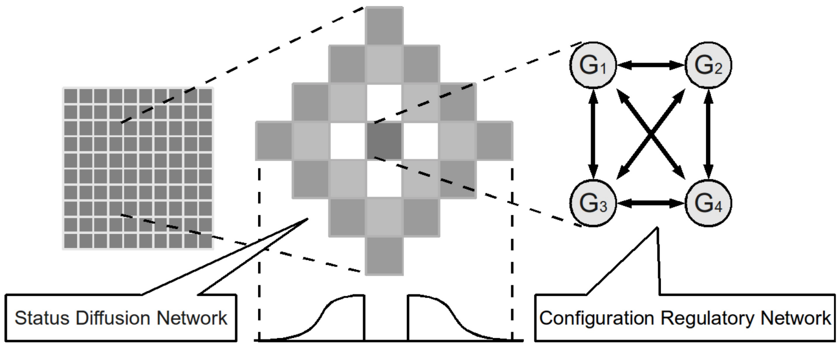

Figure 9: Two types of networks in the SABRE array

transmission amplitude (ta). A T_{ID} that is received by a processor, in addition to being interacted with, is also transmitted to adjacent cells; importantly when this transmission occurs the ta is reduced. Hence, there is a diffusion gradient for a processors transmitted T_{ID} (figure 9). The distance over which a T_{ID} diffuses is termed its zone of influence, and is determined by its initial ta, it is a key component in determining network behaviour.

When a processor receives a T_{ID} it adds to the 'concentration' (total ta) of that T_{ID}, and the resultant behaviour of its CRN is dependent on the concentration of the T_{ID}s it has received. Concentration thresholds are used to determine if one of the tasks may be expressed, or if the processor will remain dormant; too high concentrations also cause dormancy to ensure sufficient DU for fault recovery. If a task is expressed, that AU begins transmitting its T_{ID}, leading to further task allocation. The result of this developmental process is the emergence of a stable solution with all the required tasks allocated.

A faulty AU will only transmit configuration status messages that signal its faulty state, and thus the system will no longer be stable, resulting in that AU's task being allocated to a DU, and returning the system to stability. It is important to note there is a temporal component to the interactions, i.e., T_{ID}s are produced at intervals, adding to their concentation in each recieving processor, and that decays over time. The consequence of this is that, even if a required task should not be expressed due to DU requirements, then the protein levels will eventually decay sufficiently for the task to be expressed anyway.

The desired behaviour of both networks has two main contributing factors: firstly, the data dependencies of each task, i.e., dependent tasks should be close to one another; secondly, ensuring there is sufficient DU for recovery from fault situations. The solution to these conflicting requirements needs to be embedded in the dynamics of the identical CRN in each SMove processor, and the way they interact as nodes in the CSN.The underlying mathematics behind the SRN generation and task allocation process has been omitted here for brevity, for the detailed description see [31].

6.1.1. Worked Example In order to better illustrate the mechanisms described in this section, we provide a worked example of a program we have instantiated on a simulation

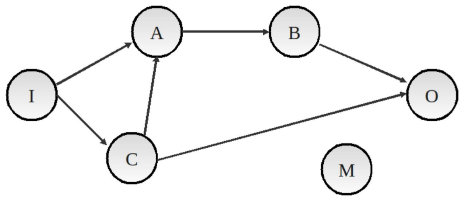

Figure 10: A partitioned task graph

M	A	▨	B	M
C	▨	▨	▨	▨
I	▨	▨	▨	O
C	▨	▨	▨	▨
M	A	▨	B	M

(a) Layout 1

M	▨	▨	▨	M
C	A	▨	B	▨
I	▨	▨	▨	O
C	A	▨	B	▨
M	▨	▨	▨	M

(b) Layout 2

M	▨	▨	▨	M
▨	C	A	▨	B
I	▨	▨	▨	O
▨	C	A	▨	B
M	▨	▨	▨	M

(c) Layout 3

Figure 11: Three stable solutions

of a 5×5 processor array. The program is partitioned into five workload-oriented tasks, which are denoted as I, O, A, B and C, and one status-oriented (immune) task, denoted as M; the data-dependancies of these processes are shown as a task-graph in figure 10. The weight of each data connection (which determines the degree of dependancy) is the same. Among them, tasks I and O, as an input and an output of the system, are pre-allocated to two processors on the edges. By optimising the parameters in the ADS for the given pre-partitioned task graph, the remaining four tasks, A, B, C and M were expected to emerge from this initial setup. The parameters defining the behaviour of the CRNs and CSN were optimised using a simple genetic algorithm (GA), in order to produce developed solutions that best matched the network criteria.

From 100 independent runs of the GA, 3 stable solutions were observed, as shown in figure 11. Except for the input and output tasks, all other tasks have at least two copies in the array; this is a result of the symmetric nature of our dynamic task allocation algorithm. Each copy of a task can be seen as a *hot back-up*, that allows the system to continue to operate while repair takes place (at a slower rate if both task copies were operating in parallel). Further, hot back-ups provide additional fault tolerance to the system: if the task on a faulty AU is not able to be reallocated, there is at least one hot back-up still working and the system can continue to operate.

We have used layout 1 (figure 11a), to demonstrate system behaviour when faults are injected. A fault is injected into a processor running task A (figure 12a). This causes the system to no longer be stable, and as a result of the current local T_{ID}s amplitudes one of the DUs in the original layout begins to run task A instead (figure 12b). A subsequent fault is injected into a processor running task B (figure 12c), and this is repaired in a similar manner (figure 12d). It is important to note that, in both cases, the desired exclusive zones of some of the processors is compromised in order for repair to occur, hence, the subsequent stable configuration of the array is dependent on the initial configuration, and the pattern and sequence of faults occuring in the array.

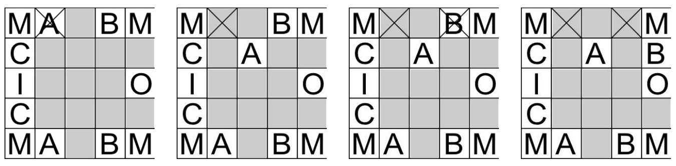

Figure 12: Intrinsic fault tolerance capability of the ADS

6.2. Artificial Immune Units

In light of the fact that when an AU is faulty at runtime the reallocation is dependent upon the configuration prior to the fault, it can be concluded that there will exist sequences of faults from which the system will be unable to recover, even though more spare resources are available to the system. This occurs as a consequence of the fact that the network parameter optimisation process only considers the initial system set-up. However, for any non-trivial system the number of possible fault sequences is very large, so if taken into consideration would make the optimisation process infeasably complex. For instance, suppose there are N processors and M AUs. AUs become faulty one after another, and at each time, a new AU will emerge to replace the failed AU. Maximally, there are N^{M-N} possible fault-recovery sequences (with length of $M-N$).

In order to address this issue, and improve the system stability, we have added an artificial immune system (AIS) to provide extra on-line assistance. This mechanism is inspired by the inflammatory response of the innate immune system [16]. During inflammation, various specific cytokines are generated to induce adult stem cells to be differentiated and replace the injured tissues. Analogously, we expect that the AIS not only actively monitor the health status of the AUs (as reported by the SMove controller in each processor), but, more importantly, the AIS will also be able to passively recognise anomalous events when an AU fails, and contribute to the recovery. Namely, the AIS should help reallocate the missing task to a DU, and avoid unrecoverable situations.

In order to do this, in addition to work-load AUs (WUs), the array also contains immune AUs (IUs), formed as part of the development process, that monitor the system and deal with exceptional situations. The operation of IUs is memory based, which is primarily inspired by the innate branch of the immune system. A memory table records the anomalous events and their corresponding responses. When an AU is faulty, if there is no replacement AU emerging and thus compensating the change of diffusion amplitudes, other AU(s), which have data dependencies with the faulty AU, will be adversely affected. An affected AU reports its current status to the IUs. The IUs receive this report and look up the anomaly-response table for a solution. If there is a corresponding entry, the IUs will cooperatively allocate the required task to a target DU (by altering T_{ID} messages within the array) and drive the system back to a new stable state. Notably, only when a DU receives more than one IUs command, it will be initiated to perform the required task.

The response table is predefined and deterministic, and thus the response is very fast. One of the simplest ways to create such a table is to inject faults into every AU sequentially and check the diffusion amplitudes every time, until an anomalous event is triggered. As discussed, when the scale of the system increases, the number of possible sequences will exponentially increase. So, a statistical sampling method, with respect to the fault probability, would be applied to reduce the computational overhead. The higher the fault probability, the higher priority that an AU has to be tested in the sequence.

It is important to note that IUs not only monitor WUs, but also themselves. This mechanism has two advantages over the conventional centralised inspection mechanism. Firstly, processors are monitored by more than one IU, and thus error detections are performed using a voting system. IUs work in a decentralised and collective way, so that they can avoid false positive diagnoses, which have lower chance to happen simultaneously to all IUs than any single one. Moreover, this mechanism prevents hierarchical inspections. There is no need to presume the monitor on top of the inspection hierarchy being fault free. In other words, when an IU fails, the system will re-create a new one, by the cooperation of the ADS and the AIS.

6.2.1. Worked Example Continuing the example from section 6.1.1, figure 13a-13d shows the final stages of a sequence of faults from which the system was unable to recover. Both of the processors performing task C become faulty in quick succession, and the network response is unable to place a replacement; additionally, as the expression of task A relies on transmission of the task C T_{ID}, those cells revert to being DUs. In figure 13e-13h, 3 of the IUs (M tasks) act to place the C task in a suitable DU, after receiving anomaly reports from the task A processors. In order to generate this response we trained the AIS with 50 randomly generated 5-fault sequences, after which 100 further random test sequences were generated to test the response of layout 1 with and without AIS support; a sequence was generated by the system choosing a random 'alive' cell to 'kill' at each time step, until the system failed. We found that even with this simplistic anomaly-response table generation method, significantly more fault sequences could be recovered from than without the AIS [31].

6.3. Hardware Overhead of Test and Repair Mechanisms

At the multi-processor layer of the hierarchy, there are four sources of hardware overhead related to the fault tolerance methods employed: the SRN function unit running on each processor, the program memory for all task configurations, the immune units, and the spare processors. The complexity of the SRN function unit is dependent on the data length of the T_{ID}s and tas that must be multiplied and the result compared; multiply and compare operations in general are not overly complex so, while we have not synthesised an SRN unit using UX cells, we believe it should require fewer cells than we envision being present in an FU. The additional SRAM required for the dynamic task allocation

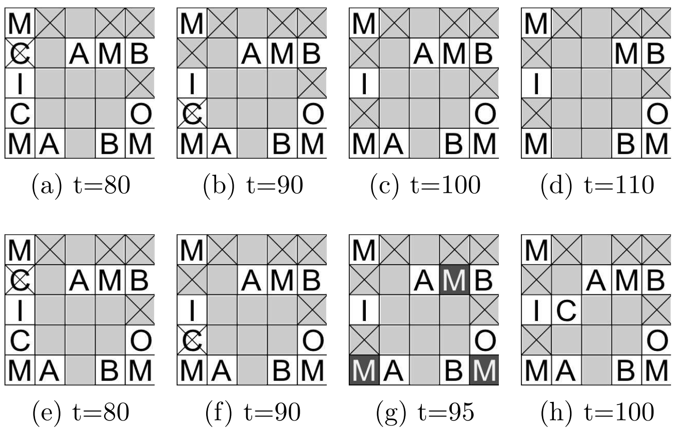

Figure 13: System response to faults, with ((a)-(d)) and without ((e)-(h)) the AIS

mechanism is dependent on the number of tasks that are present in the system, but as SRAM is relatively cheap in terms of manufacturing cost this does not represent a problematic overhead; particularly as a typical SMove processor configuration is likely to require less than 128Kb of memory.

The percentage overhead incurred by the immune units is dependent on their range of influence, and hence the density that is required to protect the task related units. Discussion of how this might best be optimised is beyond the scope of this paper. However, as is evident from the worked examples, a low density is sufficient (in the case of the examples incurring a 16% overhead).

The number of spare processors that are required by a design is dependent on the failure rate of the individual processors. Assuming a relatively low failure rate due to the inherent fault tolerance of SMove processors composed of UX cells, a low ratio of dormant to active units should be required (much lower than that shown in the worked examples). In the future we plan to analyse the failure rate of different tasks on SMove processors, so as to calculate the required number of spare processors.

7. Methods for Fault Detection in the Test and Repair Systems

An obvious issue with any fault detection and repair system is how to deal with faults within that system (referred to from hereon as secondary faults). For example, in biological systems, faults in DNA repair and the immune system can have serious consequences, such as premature apoptosis (cell death) or cancer [8]. Some secondary fault protection is inherent in the SABRE system as described here, and in the future we will be investigating other techniques.

At the top level of our fault tolerance hierarchy, inspired by the operation of the human immune system, the immune units protect each other as well as the operation of the other functional units. Further, by requiring signals from multiple immune units to trigger reconfiguration (akin to build up of chemical concentrations in a biological system), the system is protected against faults in a single unit causing harmful system reactions. This high level protection could be extended by enabling the immune units to perform pattern recognition on the functional outputs of the active processors (in addition to the configuration status messages currently monitored); thus, the task of an SMove processor that appears faulty could be reassigned to a spare unit, and the damaged unit could run self diagnostic routines to try to identify (and possibly repair) the fault. Integrity of the high level protection system is tested as it is implemented using UX cells in the SMove controllers, i.e., the low level mechanisms ensure the high level is operational and vice-versa.

An important part of the SABRE system, for which we have not developed a bio-inspired protection mechanism is the SRAM that stores the program instructions, and configuration information in each SMove controller. There exist a number of fault detection methods, with a low hardware overhead, that might be applied to ensure the integrity of this vital resource, for example Wang et al. [32] suggest a BIST circuitry that requires less than a 3% overhead; a low overhead implies that there will be a reduced chance of errors due to the lower surface area for radiation impacts. Further, as all processors contain the same memory, faulty data could be restored from another processor. Alternatively memory may be implemented using radiation hardened techniques, e.g. using MRAM [33].

Another key component of SABRE for which we have not designed a biologically protection mechanism, is the SMove controller (that also contains the UX controller). While the fact that it represents a negligible amount of hardware overhead compared to the other functional components in an SMove processor, significantly reduces its chance of error, it is vital that it remains fault free to safeguard the whole system. We already employ TMR in the UXC and so it seems reasonable to suggest doing so for the SMove related parts of the controller would not incur an enormous amount of overhead. In TMR the voting circuitry must be ensured to be error free, Kim et al. suggest using a radiation hardened voter [34], while a self-checking voter is suggested in [35]. Applying such techniques to the whole circuit can be prohibitive in terms of cost and hardware overhead (an area overhead of 33.1% using 100nm technology was reported in [36] for radiation hard logic gates), but as the controller is a small component of the system and a voter is very small compared to the controller, doing so should not incur a lot of overhead. Similarly, in the SMove ports we can employ TMR with self-checking voters, and a traditional fault tolerance approach for the parity checkers such as self-checking parity checkers [37].

The hardest area in which to apply fault-tolerance is the mechanisms within each UX cell. Faults in this system can be considered as two types: those that generate false positives (reporting a fault when there is none), and those that generate false negatives (registering fault-free when there is a fault). Faults that generate false positives, while they may be considered as triggering premature apoptosis, do not actually present a problem, as cells with faulty BIST mechanisms are equally detrimental to the system as those where the fault occurs in the functional circuitry. While false negative faults only represent a very small percentage of faults that may occur, requiring two faults in separate parts of the cell (what percentage they actually represent is beyond the scope of this paper), they are extremely detrimental to system operation. The fault tolerance mechanisms we currently employ are TMR for the ICC with a self-checking voter, and separate parity checkers for each half of the genome (such that the BIST algorithm verifies their operation). In order to protect the comparators used in BIST we could triplicate them with triplicated voters to connect them to the triplicated ICC, or use radiation hard logic gates as suggested in [36]. In any case, the comparator and voter circuits are significantly smaller than the functional circuitry they are protecting, so are less likely to be faulty.

Evaluating the efficiency of different techniques for each area of our system that biologically inspired protection mechanisms are not suitable, and investigating additional biological mechanisms will be the subject of future work.

8. Conclusions and Further Work

In this article we have described SABRE, a biologically-inspired, fault-tolerant, reconfigurable electronic architecture. The underlying structure of the system is, as in biological immune systems, hierarchical, with fault tolerance beginning with the primitive constituents, the cells. However, despite the cellular basis for our system, the divisions of our hierarchy are determined by the capabilities and requirements of the electronics, rather than those found in biology. Further, different biological immune inspired mechanisms have been employed at each layer of the hierarchy, in a bottom-up approach to artificial immune system creation; previous work has utilised a more top-down approach in an effort to more closely match the features of a single biological system [2][7]. This bottom-up approach has enabled us to utilise the most suitable mechanisms at each level of granularity, and provide a high level of fault-tolerance with a lower hardware and software overhead than heretofore.

The hierarchical cellular architecture of the SABRE system gives a number of advantages over more traditional complex electronic systems. These advantages stem from the fact that a fault in one cell (UX cell, SMove function unit, or SMove processor) is dealt with by the system in which it is contained. Hence, an irreparable fault in a cell results in it being excised from its system and replaced, with a minimal amount of backup resources being utilised, and minimal impact on the system as a whole. Furthermore, the hierarchy allows faults of increasing severity to be handled appropriately, i.e., with a greater amount of hardware and computational resources.

The base level of SABRE, Unitronics cells, are a novel prokaryote inspired cell structure, employing mechanisms inspired by the immune-like behaviours of colonies of prokaryote cells. The BIST mechanism operating within each cell, combined with fault recovery algorithms, ensure that faults in both the hardware and cell configuration data are detected and recovered from, without erroneous data being sent to the outputs of any cell. When errors are detected in a cell's configuration data (genes), it is reloaded, while detection of hardware faults causes the cell to be bypassed, and its role undertaken by spare cells in the cell array. Additionally, our novel gene-compression method means that sufficient redundant information is stored within an array of cells to facilitate recovery, without the need to store prohibitively large amounts of data in each cell.

At the second level of SABRE, we have utilised a transport-triggered architecture, SMove, whereby the reconfigurable function units in the TTA are comprised of arrays of Unitronics cells. This architecture is highly suited to partial-dynamic reconfiguration, and our extension of it, the *reclaim algorithm*, as an additional layer of fault tolerance. Each FU has a fixed number of cells, and can therefore tolerate a number of hardware faults (which use up the spare cells in the FU) dependent on the size of the task instantiated upon it. When an FU exhausts its spare cells, the *reclaim algorithm* is used to test whether other configurations can run on the damaged FU without being affected by the faults, and thus increase the number of faults that can be tolerated. Additionally, the repeated testing of an FU that was detected as being faulty gives additional protection against transitory faults that occurred during a previous test cycle.

At the top level of SABRE is a dynamic task allocation algorithm operating on an array of SMove processors, the operation of which is inspired by the developmental process and innate-immune system in biological systems. Task allocation is implemented using a network of artificial *protein* interactions between processing networks operating in each SMove processor. It is ensured that some processors remain dormant to act as back-ups, to which tasks are dynamically assigned when a processor fails. In addition, some processors are assigned immune tasks, and these monitor the other processors to aid recovery from fault situations which the array is unable to recover from.

In future work we intend to further investigate the idea of training an immune response to operate at the different levels, i.e., providing an acquired immune system; additionally, such a system would be able to identify erroneous processor behaviour, and learn successful responses to faults it encounters during operation. At the SMove processor level, it will be investigated whether fault signatures, created by recording the erroneous results from a damaged FU, can usefully be associated with solutions so that the correct solution can be more quickly recalled. It may be possible that faults with similar signatures have the same solution, and hence the immune system would respond more swiftly to faults similar to those seen before. At the top level we will investigate whether different training regimes for the development network and AIS can be used to create a more robust system. At both these hierarchical levels, we will investigate whether an AIS with associative memory (or similar pattern matching algorithm) might allow an AIS to be trained with a few representative fault examples and provide an improved immune response, i.e., a form of artificial vaccination.

Another key area for future work is to investigate biologically inspired approaches for the areas of the design where we currently rely on more traditional ones. In particular, fault-tolerance mechanisms in the test and repair systems, where we rely in large part on TMR with self-checking voters.

9. Acknowledgments

This research work is supported by the Engineering and Physical Sciences Research Council of the United Kingdom under Grant Number EP/F062192/1 and EP/F067623/1.

References

[1] T.-T.Y. Lin and D.P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4), oct 1990.

[2] C. Ortega, D. Mange, S. Smith, and A.M. Tyrrell. Embryonics: A bio-inspired cellular architecture with fault-tolerant properties. *Genetic Programming and Evolvable Machines*, 1(3):187–215, 2000.

[3] X. Zhang, G. Dragffy, and A.G. Pipe. Bio-inspired reconfigurable architecture for reliable systems. In *VLSI'03: The 2003 International Conference on VLSI*, pages 34–40, 2003.

[4] X. Zhang, G. Dragffy, and A. G. Pipe. Embryonics: A path to artificial life? *Journal of Artificial Life*, 12(3):313–332, 2006.

[5] Y. Thoma, G. Tempesti, E. Sanchez, and J. M. Arostegui. Poetic: an electronic tissue for bio-inspired cellular applications. *BioSystems*, pages 191–200, 2004.

[6] L. Prodan, G. Tempesti, D. Mange, and A. Stauffer. Biology meets electronics: The path to a bio-inspired fpga. In *Proceedings of 3th International Conference on Evolvable Hardware: From Biology to Hardware*, pages 187–196, 2000.

[7] J. Rossier, Y. Thoma, P.A. Mudry, and G. Tempesti. Move processors that self-replicate and differentiate. In *Int. Workshop on Biologically-Inspired Approaches to Advanced Information Technology*, 2006.

[8] M. Jones, R. Fosbery, and D. Taylor. *Biology.* Cambridge University Press, 2000.

[9] P. E. Greenberg. Tiny teamwork. *Nature*, 2003.

[10] M. A. Suchard. Stochastic models for horizontal gene transfer: Taking a random walk through tree space. *Genetics*, 170(1):419–431, 2005.

[11] H. Corporaal. *Microprocessor Architectures : From VLIW to TTA.* John Wiley & Sons, 1997.

[12] P.A. Mudry. *A hardware-software codesign framework for cellular computing.* PhD thesis, EPFL, 2009.

[13] G. M. Edelman and J. A. Gally. Degeneracy and complexity in biological systems. *Proceedings of the National Academy of Sciences of the United States of America*, 98(24):13763–13768, 2001.

[14] J. Gerhart and M. Kirschner. *Cells, embryos, and evolution: toward a cellular and developmental understanding of phenotypic variation and evolutionary adaptability.* Blackwell Science, 1997.

[15] I. R. Cohen. *Tending Adam's garden: evolving the cognitive immune self.* Elsevier Academic Press, 2000.

[16] A. Abbas, A. Lichtman, and S. Pillai. *Cellular and Molecular Immunology, 6th Ed.* Saunders Elsevier, 2007.

[17] T. G. W. Gordon and P. J. Bentley. Development brings scalability to hardware evolution. In *NASA/DoD Conference on Evolvable Hardware, IEEE Computer Society*, pages 272–279, 2005.

[18] M. A. Trefzer, T. Kuyucu, J. F. Miller, and A. M. Tyrrell. A model for intrinsic artificial development featuring structural feedback and emergent growth. In *IEEE Congress on Evolutionary Computation*, pages 301–308, 2009.

[19] M. Samie, G. Draggfy, A. Popescu, A. G. Pipe, and J. Kiely. Prokaryotic bio-inspired system. In *NASA/ESA Conference on Adaptive Hardware and Systems*, pages 171–178, 2009.

[20] M. Samie, G. Dragffy, A. M. Tyrrell, A. G. Pipe, and P. Bremner. A novel bio-inspired approach for fault-tolerant vlsi systems. *IEEE Transactions on VLSI*, 2012. In press.

[21] M. Samie, G. Dragffy, and T. Pipe. Unitronics: A novel bio-inspired fault tolerant cellular system. In *NASA/ESA Conference on Adaptive Hardware and Systems*, pages 58–65, 2011.

[22] G.L. Smith and L. de la Torre. Techniques to enable fpga based reconfigurable fault tolerant space computing. In *Aerospace Conference, 2006 IEEE*, page 11 pp., 2006.

[23] H. H. Hana and B. W. Johnson. Concurrent error detection in vlsi circuits using time redundancy. In *IEEE Southeastcon '86 Regional CEnf.*, pages 208–212, 1986.

[24] J. Pate1 and L. Fung. Concurrent error detection in alus by recomputing with shifted operands. *Computers, IEEE Transactions on*, C-31(7):589–595, 1982.

[25] Sandi Habinc. Functional triple modular redundancy (ftmr): Vhdl design methodology for redundancy in combinatorial and sequential logic. Technical report, NASA, 2002.

[26] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, pages 59–65, 2009.

[27] V. Braitenberg. *Vehicles: experiments in synthetic psychology.* MIT Press, 1986.

[28] P. Bremner, M. Samie, G. Dragffy, A. G. Pipe, and Y Liu. Evolving cell array configurations using CGP. In *Proceedings of the 14th European Conference on Genetic Programming.*, pages 73–84, 2011.

[29] P. Bremner, M. Samie, A. G. Pipe, J. A. Walker, and A. M. Tyrrell. Multi-objective optimisation of cell-array circuit evolution. In *Proceedings of 11th. Congress on Evolutionary Computation*, pages 440 – 446, 2011.

[30] P. C. Haddow. Evolvable hardware: A tool for reverse engineering of biological systems. In *International Conference on Evolvable Systems: From Biology to Hardware*, pages 342–351, 2008.

[31] Y. Liu, J. Timmis, O. Qadir, G. Tempesti, and A. M. Tyrrell. A developmental and immune-inspired dynamic task allocation algorithm for microprocessor array systems. In *Artificial Immune Systems, 9th International Conference, ICARIS 2010*, pages 199–212, 2010.

[32] Chih-Wea Wang, Chi-Feng Wu, Jin-Fu Li, Cheng-Wen Wu, T. Teng, K. Chiu, and Hsiao-Ping Lin. A built-in self-test and self-diagnosis scheme for embedded sram. In *Test Symposium, 2000. (ATS 2000). Proceedings of the Ninth Asian*, pages 45–50, 2000.

[33] Johan Akerman. Toward a universal memory. *Science*, 308(5721):508–510, 2005.

[34] Hyunki Kim, Hyeuntae Lee, and Keyseo Lee. The design and analysis of avtmr (all voting triple modular redundancy) and duplex system. *Reliability Engineering and System Safety*, 88(3):291 – 300, 2005.

[35] J.M. Cazeaux, D. Rossi, and C. Metra. New high speed cmos self-checking voter. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, pages 58 – 63, july 2004.

[36] Quming Zhou and K. Mohanram. Gate sizing to radiation harden combinational logic. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(1):155 – 166, 2006.

[37] Dimitris Nikolos. Optimal self-testing embedded parity checkers. *IEEE TCOMP*, 47:313–321, 1998.