Application-Aware Optimization of Redundant Resources for the Reconfigurable Self-Healing eDNA Hardware Architecture

Michael Reibel Boesen, Jan Madsen, Paul Pop
Technical University of Denmark, DTU Informatics
{mrb,jan,pop@imm.dtu.dk}

Abstract

In this paper we are interested in the mapping of embedded applications on a dynamically reconfigurable self-healing hardware architecture known as the eDNA (electronic DNA) architecture. The architecture consists of an array of cells interconnected through a 2D-mesh topology. Each cell consists of a processor and an Arithmetic Logic Unit (ALU). Applications are modeled as task graphs. We propose a Tabu Search-based approach for the mapping of an application to the reconfigurable architecture, such that the performance is maximized. When faults occur, the self-healing moves the affected functionality to spare-cells. We optimize the number and placement of spare-cells such that the performance overhead is minimized in the fault-free scenario and the application degrades gracefully in case of faults. This has been done using three different spare-cell placement strategies. We use Monte Carlo simulation to determine the average performance overhead increase due to fault occurrences. The approach has been evaluated using a large number of benchmarks and have shown that the performance loss is reduced with 16% for the best spare-cell placement strategy.

1. Introduction

There has been a lot of research on mapping applications onto MultiProcessor System-on-Chips (MPSoCs) [4,7]. However, due to the high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels, the number of transient and permanent faults has increased considerably [8]. Researchers have used costly hardware redundancy to tolerate failures in processing elements [10]. To reduce hardware costs, two fault-tolerance approaches have been proposed: time-redundancy, where the functionality is re-executed to tolerate a transient fault [9], and spatial-redundancy, such as active replication [13]. Few researchers have studied the performance impact of redundancy on the application [16]. In [16] the authors use checkpointing and active replication to synthesize fault-tolerant hard real-time systems.

In this paper we are concerned with multiple, potentially simultaneous, permanent faults. Many of the current approaches to tolerating permanent faults have the drawback that they use massive amounts of redundancy to allow recovery without explicit error detection (i.e., fault ``masking"). One solution is to use error detection followed by task migration, where the tasks are moved to a healthy processing element. However, task migration incurs a large overhead in MPSoCs [3], due to the high amount of data that needs to be transferred between the individual processing nodes.

To solve the problem of fault-tolerance in hardware, several researchers have considered bio-inspired reconfigurable architectures such as [11,14]. In this paper we consider a bio-inspired dynamically reconfigurable hardware architecture (named eDNA (for electronic DNA) [1,17-19] capable of self-organization and self-healing (patent pending, however, the work in this paper is not only applicable to this architecture).

We are interested to optimize the number and placement of redundant spare-RUs (reconfigurable units) such that the permanent faults are tolerated and the performance overhead of the application is minimized, even in case of multiple permanent faults. We have proposed an approach for the mapping of an application to the architecture and the distribution of spare-RUs, such that the performance is maximized. The approach is based on a Tabu Search [5]. We use Monte Carlo simulation to determine, for a given number of spare-RUs and their placement, the average performance overhead increase due to fault occurrences and consequently, self-healing.

The paper is organized as follows. Section 2 introduces our eDNA architecture. Section 3 describes the model of the eDNA architecture and in particular the architecture, application and fault model used. Section 4 introduces the problem as well as discusses the redundancy optimization approach, section 5 explains details about the optimization strategy used, section 6 presents and discusses the experimental results and finally, the conclusion is presented in section 7.

2. The eDNA Concept

Figure 1 shows an overview of the eDNA architecture which consists of a *distributed* array of multiple *homogenous* processing units known as *electronic cells* (cells). The application to be executed on the architecture, called the eDNA program, is programmed using the eDNA programming language [1,17,18]. The compiled binary version of the eDNA program, is fed to *all* cells of the architecture. Each cell stores a copy of the binary version in a local RAM block. The compiled version of the eDNA program divides the program into tasks, which are considered as *genes* of the DNA. When a cell has to be configured, it finds its corresponding gene from the stored eDNA program and configures the cell into the function (or task) given by the corresponding gene – this process is known as self-organization. In terms of logical granularity, one gene is equivalent to one arithmetic operation of the eDNA program.

Each cell contains a configuring unit and a computing unit. The task of the configuring unit is to identify the gene of the cell and to configure the computing unit to carry out this function. In our prototype implementation the computing unit is a 32-bit ALU and the configuring unit is a Xilinx PicoBlaze [6], which is an 8-bit VHDL synthesizable soft-core provided by Xilinx. The PicoBlaze is only responsible for executing the *ribosomal DNA program* (referring to the intracellular organelle in biological cells, responsible for synthesizing proteins and consequently functionality of the cells). This program performs the *self-organizing* and *self-healing* algorithms of the eDNA architecture. All cells contain a copy of this program. At runtime when no faults occur a 32-bit ALU is used for data processing, where it performs an operation on the two 32-bit operands A and B. A detailed block diagram of a cell can be seen in figure 2.

Observe that *no* centralized processing unit is present. The cells complete the self-organizing and self-healing completely autonomous.

The cells communicate with each other through a 2D-mesh-8 architecture, where each cell communicates with at most 8 adjacent neighbors depending on position. We assume that the packet transfer in the network is implemented with a fault-tolerant handshaking protocol, which is capable of routing around faulty links, such as [20].

2.1 Self-organization: Gene Mapping and Placement

Self-organization is about interpreting the eDNA program and herby creating a physical mapping between cells and the tasks of the program. When all cells have received the complete eDNA program the self-organization begins. During self-organization each cell locates its gene and configures the 32-bit ALU accordingly, by moving the particular genes to a block RAM shown in figure 2 as the Gene RAM. Each cell locates its gene by using its *cell ID*, which identifies its position in the array. The cell ID is an integer from 0 to N-1 (where N is the number of cells) The cell ID is distributed together with the eDNA and is consequently, responsible for mapping the functionality of the eDNA program onto the different cells. The self-organization algorithm works by counting the number of genes and comparing it to the cell ID of this eCell. When the number of genes equals the cell ID the gene of this eCell is found and self-organization is complete.

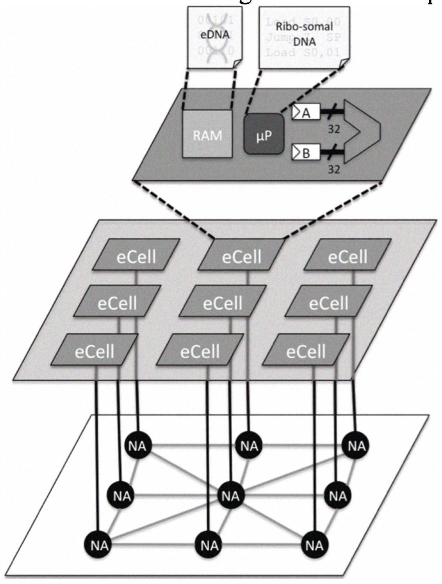

Figure 1 Overview of the eDNA architecture.

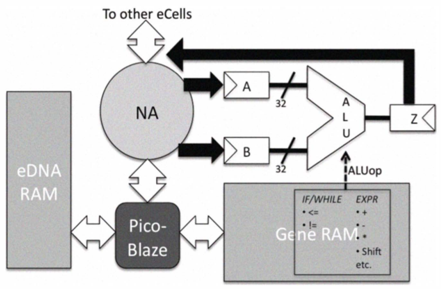

Figure 2 cell Block Diagram.

2.2 Self-healing

A crucial part of the self-healing process is the fault-detection. However, it is not important for this paper and has been left out due to space limitations. Details about it can be found in [18]. It is however, important to realize that the cells cooperatively test each other – therefore fault-detection and self-healing of a faulty cell is done by another cell than the one, which is deemed faulty.

When a fault is detected the application is paused and the self-healing algorithm is activated, which consists of the following steps:

1. Restore the gene of the faulty cell at a spare cell.
2. Restore the gene state at the spare cell

When both steps are completed the application resumes. Step 1 is the restoration of the functionality of the faulty cell at a spare cell. It is completed by remapping the cell ID of the faulty cell to a spare cell and then broadcast a message to all cells requesting them to run the self-organizing algorithm again. This will cause the non-faulty cells to speak with the spare cell instead of the faulty cell and will cause the spare cell to realize that it is now an active cell and participating in running the application.

Step 2: The gene state is basically the two registers (A and B of figure 2) holding the operands of the cells ALU. Recovering the gene state is quite complex and is not the scope of the paper, but information about it can be found in [18]. An example of the self-healing can be found in section 3.5, where we formally describe how the self-healing is performed.

More information about the eDNA architecture can be found in [1,17-19]

3. eDNA System Model

This section will describe the model of the eDNA architecture, used in the optimization approach. Figure 3 shows the system model of the eDNA architecture.

3.1 Architecture model

The eDNA architecture *H* consists of a *distributed* array of *N homogenous* cells, which are interconnected through a 2D mesh topology. The homogenous cells can be assigned one of three different roles (as seen in figure 3): (1) **func-cell** (depicted in dark-gray) implement a part of the application; (2) **spare-cell** (depicted in black) are at the time of configuration idle, but in case of a fault in another cell, it can be configured to take the function of the faulty cell; (3) **I/O-cell** (depicted in light-gray) are located along the left and right edges of the chip and they do not perform any part of the application and can be viewed as I/O pins. Furthermore, observe in figure 3 that each cell $C_i \in N$ is uniquely identified by an integer *i*, called the *cell-ID*.

3.2 Application model

The applications can be modeled as a directed acyclic task-graph $G=(V,E)$. Each node $\tau_i \in V$ represents a task, which implements part of the functionality of the application. An edge $e_{i,j}$ between τ_i and τ_j means that τ_j receives an input from τ_i. The tasks can start when all their inputs have arrived and issue all the outputs when they terminate. An example of a task-graph representation of an application is depicted in figure 3. The task graph *G* is polar i.e., there is a source node τ_{source} and a sink node τ_{sink} (which can be added as dummy nodes (node D on figure 3) if they do not exist). The end-to-end delay δ_G of the graph *G* is determined by the finishing time of τ_{sink}. Note, that δ_G is given by the length of the critical path.

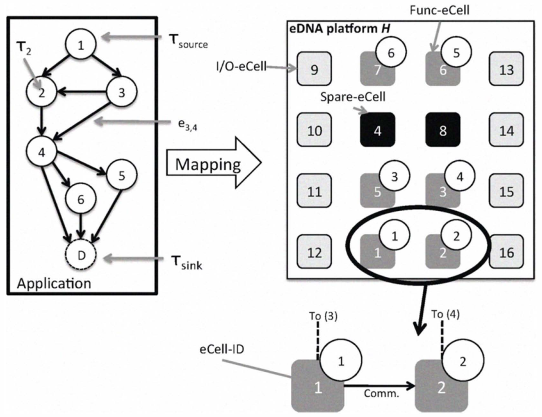

Figure 3 System model example and self-organization.

3.3 Self-organization

The functionality of each task $\tau_i \in V$ is *mapped* to a func-cell denoted by $M(\tau_i)$. This mapping is stored in a mapping table in *all* cells. All cells will have access to the functionality of all tasks. The memory (used to store the tasks information) is protected by data integrity tests, such as parity codes, and can be considered fault-tolerant. Hence, if a func-cell implementing τ_i fails, the other func-cells will have a copy of the task, which will be implemented in a spare-cell after self-healing (see section 3.5).

Each task τ_i also contains information about which tasks are connected to τ_i, i.e., it contains all outgoing edges of the type $e_{i,j}$. This information can be used to search the network for the cell implementing τ_j and consequently, a communication path can be set up. Given (1) an edge $e_{i,j} \in E$, (2) the mapping of τ_i and τ_j, (3) the amount of data communicated between τ_i and τ_j and (4) a given routing algorithm, we can determine the communication cost $c_{i,j}$.

Note the difference between self-organization and mapping. The *mapping* is performed offline and describes which func-cell each task is mapped to. The *self-organization* is the process where each task interprets the mapping and implements the task.

3.4 Fault model

We are interested in tolerating up to *k* permanent faults in func and spare-cells – I/O cells are ignored in this study.

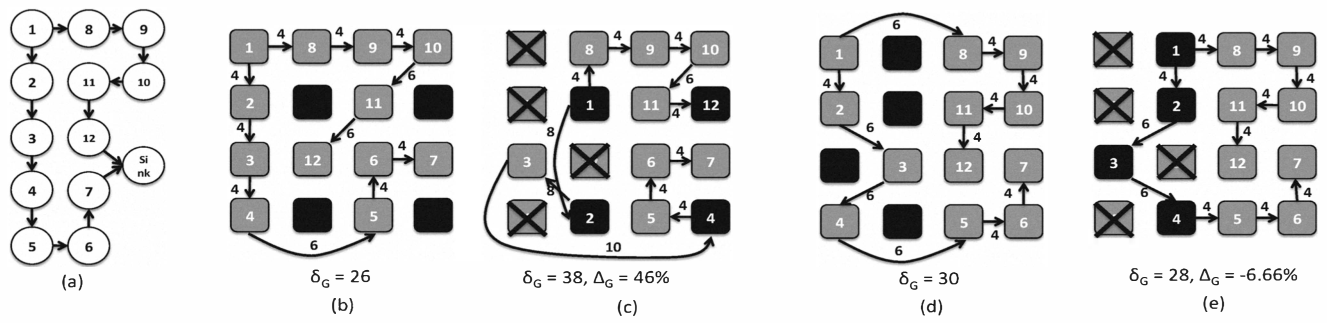

Figure 4 Placement strategies (a) (UD) the task-graph, (b) (UD) initial mapping, (c) (UD) self-healing, (d) Naive fault specific mapping 0 faults, (e) Naive fault specific mapping 4 faults.

The value of k depends on the particular architecture and application and is given by the developer. The fault model is in this case focused on permanent faults, but it could be extended to transient faults as well by adding a periodic checking of failed func-cells to see if the fault detected was a transient one. Observe that we in the model only consider permanent faults in the func-cells.

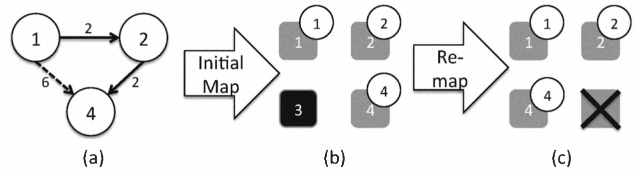

Figure 5 Self-healing example: (a) the task-graph, (b) initial mapping, (c) self-healing.

3.5 Self-healing

If a fault is detected, the procedure known as self-healing is activated. The first step is to find the nearest spare-cell, which should take over the functionality of the failed cell. Local memory of the cell contains a list of spare-cells; their cell-ID and placement as mentioned in 3.3. Each cell implementation contains the functionality to compute the Manhattan distances between two cells. Based on this metric, the nearest spare-cell C_{spare} can be selected. In figure 5 we have the task graph from figure 5(a) mapped to the architecture in figure 5(b). C_4 suffers a permanent fault, which triggers the self-healing process. The nearest spare-cell is calculated to be C_3. In the next step, C_{spare} (in the example, C_3) is labeled with the cell-ID of the faulty func-cell (in the example, C_4), and the self-organization process (described section 3.3) is invoked. C_{spare} will now implement the failed functionality (to which it has access, in the form of the task-graph and which in figure 5 is τ_4) and all cells are notified of this new mapping and consequently update the mapping tables. This means that once the self-healing is done, we will have a new *physical* mapping of the application, i.e., a task has been moved, but the mapping between task and cell-ID is still the same. Note, that the new mapping may result in a different $c_{3,4}$ depending on, which spare-cell is used for self-healing. This might lead to an increase of δ_G due to the self-healing, captured by Δ_G, which is termed the *overhead increase*. Self-healing is similar, but not identical to, task migration. The self-healing is a simpler form of task migration where, for instance, it is not necessary to *physically* move the task, since all cells have access to all tasks.

4. Problem Formulation

The problem we are addressing in this paper can be formulated as follows: Given (1) an eDNA architecture with func- and I/O-cells, $N_{cell} \cup N_{I/O}$ (2) an application G and (3) the number k of permanent faults to be tolerated, we are interested to determine (i) the number $|N_{spare} \geq k|$ spare-cells, (ii) the placement $P(C_i)=(x,y)$, where (x,y) represents the 2D location of each spare-cell C_i and (iii) the mapping $M(\tau_i)$ of each task $\tau_i \in V$ in the application such that the end-to-end delay δ_G of the application is minimized in the fault-free scenario, and (iv) the average Δ_G, due to self-healing is minimized in case of faults, i.e., the application degrades gracefully.

We will in the following show that there are several strategies with which to solve all of the four sub goals.

4.1 Spare-cell placement strategies

Consider $k=4$ to be tolerated, figure 4(b) shows a mapping of the application G from figure 4(a) on an architecture with 4 spare-cells. Note that the numbers in the boxes in the figure refer to the task-numbers seen on figure 4(a). (For the sake of simplicity, we have removed the I/O-cells.) In the eDNA architecture communication between neighboring cells takes 3-6 times longer than the execution of a task. Therefore in our model we also assume this relationship and consequently, ignore the execution time of the individual tasks. The end-to-end delay δ_G for figure 4(b) is 26. The mapping has been done such that δ_G is minimized. In the example (figure 4 (b)) *4* spare-cells have been placed uniformly on the architecture. This is the simplest form of placement denoted with uniform distribution (UD). Considering this mapping and XY-routing, the communication cost that results is depicted between each communicating cell. However, such a placement will lead to a large reduction in performance in case of faults. Let us consider figure 4(c), where we have 4 permanent faults, depicted with an ``X''. The self-healing will react to these faults by moving the affected functionality to the new mapping configuration as depicted in figure 4(c). This new mapping will result in a performance degradation to δ_G = 38 (and consequently a Δ_G = 46%), due to the increased communication cost needed between the spare-cells. Introducing more spare-cells, beyond what is required for tolerating 4 faults, such that the average distance to a spare-cell is minimized, can reduce these communication costs. This, however, will increase the cost of the architecture (i.e. more cells equals more power and area). Another solution, depicted in figure 4(d) is to place the spare-cells such that the communication cost in case of faults are minimized (figure 4 (e)). However, (i) in this case the δ_G of the application assuming no faults will increase to 30 and (ii) it may not be a good placement for a different fault scenario. The challenge is to find a placement of spare-cells, which obtains a low delay in case of no faults and which produces the smallest increase for any fault scenario of *k*-faults.

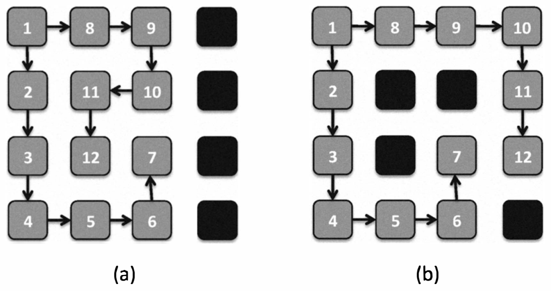

Figure 6 (a) initial mapping 0FA, (b) initial mapping MD d=1

If we allow the moving of spare-cells from the uniform configuration, the best solution for the fault-free scenario is where the spare-cells are moved away from the communication paths of the application as depicted in figure 6(a) and named *0-faults-anticipated* (0FA). The figure clearly shows that 0FA utilizes the freedom of spare-cell placement to the fullest by moving all spare-cell out along the edges. Furthermore, the critical path is also very compact, which will reduce the end-to-end delay δ_G. The compactness also explains the bad performance when faults are introduced, because many of the func-cells on the critical path share the same nearest spare-cell. Which means that once multiple faults in the critical path occur they will increase the delay dramatically.

The third approach is a combination of UD and 0FA, where we allow the movement of spare-cells from their initial uniform configuration out of the communication paths of the application, but add a constraint which ensures that all cells as a minimum have one spare-cell within a radius of *d* cells. Figure 6(b) shows an example of this, where *d=1*. This approach is called minimum spare-cell distance (MD). MD will not spread the spare-cells out too much (compared to 0FA). However, MD still has enough freedom to place the critical path tasks close to each other. This should reduce δ_G for the fault free scenario and once faults are introduced all func-cells are quite close to a spare-cell and consequently the Δ_G overhead increase will not be as large as with 0FA on average.

5. Optimization strategy

Our optimization strategy consists of three steps:

(1) In the first step we decide the initial number $|N_{spare}|$ of spare-cells and place them uniformly on the architecture. The initial number of spares is such that $|N_{spare}| \geq k$ and the distance between spare-cells is not too large.

(2) In the second step we decide the mapping of the application G on N_{cell} such that the end-to-end delay δ_G or the application completion time is minimized for the *fault-free scenario*. We find the mapping using a Tabu Search (TS) optimization metaheuristic, which minimizes δ_G of the application G (see section 5.1). TS considers the placement approaches: 0FA (figure 6 (a)), UD (figure 4 (b)) and MD (figure 6 (b)) with different radiuses *d*. This means that all tasks might be relocated. Note that this may also include relocating spare-cells.

(3) In the third step we evaluate the robustness of the system (architecture and mapping of application) to permanent faults, given a mapping M and a placement P of the $|N_{spare}|$ spare-cells produced in step (2). This is time consuming, since it has to consider a large number of possible fault scenarios, consequently it cannot be performed inside the TS. Therefore the evaluation will be performed using Monte Carlo Simulation (MCS) [15]. The MCS will introduce faults in random cells, which then will cause the architecture to self-heal and consequently change the mapping resulting in an increased δ_G. The MCS will perform 10,000 iterations as follows: (1) Select between 0 to k random cells and introduce a permanent fault in each of them; (2) invoke self-healing to move the failed functionality resulting in a new mapping and (3) for each simulation find and record the δ_G increase. This increase will be averaged over all simulations, denoted with Δ_G. The result is the average increases in δ_G after self-healing due to permanent faults, captured by Δ_G. The designer will iterate several times through these steps using different values for $|N_{spare}|$ until the desired implementation is obtained, which has a good performance in the fault-free scenario (δ_G) and degrades gracefully in case of permanent faults (Δ_G).

5.1 Cost function

Inside the TS we use the critical path CP_G of graph G as a cost function. This will give us the end-to-end delay δ_G of the application. For each task τ_i we know the execution time E_i of a cell (because they are homogenous). Considering a mapping M and the XY-routing employed, we can determine the communication costs $c_{i,j}$. The simplest way of doing it would be to just calculate the distance in number of hops from τ_i to τ_j. However, this is imprecise because it will not take the queuing time (packets waiting to be routed due to congestion) into account. Instead, we will calculate the communication cost $c_{i,j}$ for an edge $e_{i,j}$ by adding: (1) the distance in number of hops from τ_i to τ_j and (2) the queuing time associated with data travelling from τ_i to τ_j. The queuing time is the time data waits in a buffer at each router because of traffic. This means that in order to calculate the queuing time it is necessary to schedule the traffic in the mesh network. For the purpose of calculating (2) we have decided to use List Scheduling (LS) [12] to get a reasonable estimate of the queuing times. (2) works in the following way. The tasks are mapped to a network model of the 2D-mesh NoC. Then the task graph is simulated on the network model in order to study the data flow and consequently compute the queuing times. We call this approach for calculating the critical path LSXY. Observe, that this cost function can be used regardless of interconnect-architecture used. Only the way the critical path is found is different.

5.2 Tabu Search-based mapping

We use Tabu Search (TS) [5] metaheuristic to decide the mapping M for the tasks and the placement P of the spare-cells. TS uses design transformations (moves) to search the solution space. The neighbors of the current solution, which can be obtained through performing the moves, are called the candidate set.

We consider a single type of move, which swaps the functionality (i.e. tasks) of two cells. We consider two types of cells for the swap: (i) cells that implement the tasks of the current application, from the set $C_i \mid M(\tau_i)=C_i, \forall \tau_i \in V$ and (ii) spare-cells, from the set $|N_{spare}|$. Thus, a move would either swap the mapping of the tasks or change the placement of a spare-cell by swapping its position with a func-cell.

The candidate set can become prohibitively large and evaluating every move using the LSXY cost function would thus take too much time. We reduce the candidate set to speed up the search without degrading the quality of the final solution. Figure 7 shows how the candidate set is reduced. Instead of evaluating every move we only evaluate the moves that might improve the solution. At each iteration of the TS, TS searches the task graph G for the critical path (as described in section 5.1) also using the placement of the different cells in the hardware architecture H. Then it starts from the root of the critical path and examines where the neighbors to this is located. In the case of figure 7(b) the cell implementing τ_2 is selected.

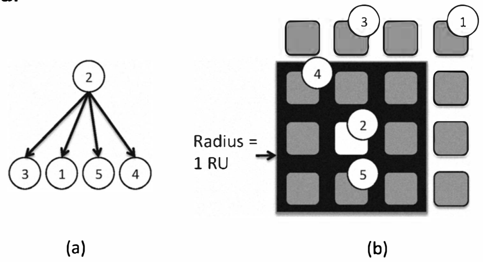

Figure 7 Example of candidate set reduction.

Figure 7 (a) shows the simple task graph as seen from this cell C_2. TS now considers only the swapping C_2 with the mapping $M(\tau_2)$ for which an edge $e_{2,j}$ or $e_{j,2}$ exists and where the Manhattan distance to it is bigger than 1. As seen in figure 7(b) this would be the cell implementing τ_3 and the cell implementing τ_1. Consequently, for this particular selection only these two cells are considered for swapping. Furthermore, TS only tries to swap with cells, which are within the radius. Consequently, we consider for a swap move only the cell implementing tasks on the critical path or tasks which are direct successors or predecessors of tasks on the critical path and which are distant enough to be interesting. Adopting this candidate set reduction leads to a big speedup of the TS algorithm compared to a straightforward approach, which simply swapped every cell with every other cell. Finally the algorithm perform the radius based test (if the MD strategy is used) and change of the current candidate solution X_{best}.

Once the best move from the candidate set is found, this solution X_{best} is placed in the tabu list (to maintain a selective history of the exploration) and the exploration continues from X_{best}. TS uses the concept of ``selective history'' of the recently visited solutions (a ``tabu'' list) to avoid visiting the previous explored solutions. However, tabu moves are also accepted if they are better than the best so far solution. If no improvement on the best so far is seen for a large number of iterations, TS employs *diversification*, where several random moves are performed to guide the search to unexplored areas. In our TS implementation, we also perform from time to time *intensification* phases. The idea is to explore more thoroughly those areas of the search space that look promising. This is achieved by fixing those tasks from the best-so-far solution that have not been involved in swaps for the recent history and concentrating only on moves that change the mapping and placement for the rest of the tasks and spare-cells, respectively.

Approach	6x4 4 spares	8x6 9 spares	10x8 16 spares
0FA	2.57%	2.71%	1.42%
UD	4.11%	5.08%	1.43%
MD(d=3)	0.00%	0.00%	3.95%
MD(d=4)	4.11%	2.22%	0.00%

Figure 8 APD values for the fault free scenario

6. Experimental Results

In the first set of experiments we are interested to compare the three Tabu Search-based (TS) mapping approaches: 0-faults-anticipated (0FA); Uniform distribution (UD) and Minimum spare-cell distance (MD). The approaches are compared on two parameters: (1) Cost of the fault free scenario δ_G and (2) Δ_G the average increase in δ_G when up to k faults are introduced. Furthermore, we will experiment with three different sized architectures: 6x4, 8x6 and 10x8. For each of the architecture we consider 10 randomly generated task-graphs with 12 to 56 tasks. All results presented here are calculated as the average percentage deviation (APD) from the best cost function obtained. The APD is calculated by (1).

$$APD = \frac{(cost_{candidate} - cost_{best}) \cdot 100\%}{cost_{best}} \qquad (1)$$

Observe, that an APD value of 0% indicates that it is the best and consequently an approach having for instance an APD of 4% means that that approach is 4% worse than the best. Figure 8 shows the APD for the three different architectures for the fault free scenario (with 4, 9 and 16 spare-cells, respectively). For the 6x4 and 8x6 architecture it is seen that MD with *d=3* is better compared to 0FA and UD by roughly 2-5% and for the 10x8 it is seen that MD with *d=4* is better by 1-4%. We would expect that 0FA would be the best in the fault free scenario because it gives the Tabu Search algorithm the highest amount of freedom. 0FA should be followed by MD and then UD. However, it turns out that MD and 0FA are almost equally good (1-2% difference). This means that the amount of freedom provided by MD is enough to obtain an initial mapping with a good performance. Also in order to give the TS enough freedom *d=3* and *4* were chosen. However, for the smaller architectures this is quite a lot compared to the width of the architecture. This is why we see that there are very little difference between 0FA and MD.

6.1 Spare-cell placement approach evaluation

In the second set of experiments we were interested to compare the three spare-cell placement approaches in terms of overhead increase of performance (Δ_G) due to fault-occurrences. Figure 9 shows the APD of Δ_G for the cases when up to 16 faults are introduced. We can see that the MD with *d=4* is the best approach when multiple faults are introduced, i.e., a performance increase of 4-17% is seen depending on the number of faults. The difference between MD (*d=3* and *d=4*) and UD is almost constant no matter how many faults occur. However, for the 0FA we see that the performance degrades very rapidly as the number of faults increased to 6. After this, it is almost constant until 11 faults, where the difference decreases. The reason for this is the non-optimal mapping performed by the self-healing process, which selects the nearest spare-cell for tolerating the fault, considering Manhattan distance (as presented in section 3.5).

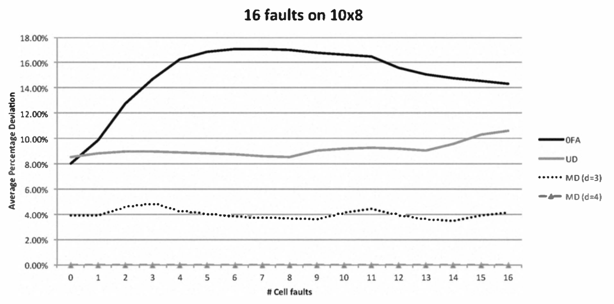

Figure 9 Comparison of 0FA, UD and MD for k=1 to 16

The message is that for mapping it is better to give TS the freedom to move the spare-cells and not constrain it by fixing the spare-cells as in the UD-approach. However, a constraint such as the radius in the MD, keeping TS from pushing the spare-cells out along the edges of the architecture is beneficial.

# spare-cells	δ_G	Δ_G
4	26	7.6%
9	23	3.5%
16	27	1.5%

Figure 10 δ_G and Δ_G values for the different architectures implementing the CORDIC algorithm.

6.2 Real example

In the final experiment we tested our approach on a real life example. Suppose we want to implement the COordinate Rotation DIgital Computer (CORDIC) algorithm on an 10x8 architecture and we want to figure out the number of spare-cells to use assuming *k=4*. Figure 10 shows the δ_G and the Δ_G values for the algorithm with 4, 9 and 16 spare-cells. The best initial mapping with a δ_G = 23 is obtained in the case when 9 spare-cells are used. The performance degrades the least (for the 4 permanent faults simulated using 10.000 MCS simulations), when 16 spare-cells are used, Δ_G = 1.5%. In this case, the best decision would be to use 9 spare-cells, which result in only a 3.5% overhead increase on average. The mapping and placement is obtained by our proposed TS implementation with the minimum spare-cell distance (MD) strategy with a radius of 4.

7. Conclusions

This paper presented an approach to optimize the redundancy introduced by the spare-cells of a bio-inspired reconfigurable self-organizing and self-healing architecture known as the eDNA architecture. We proposed a tabu-search based metaheuristic to maximize the performance of the architecture in the fault-free scenario by optimizing the mapping of functionality to the func-cells. When faults are introduced the performance will degrade due to increased communication delays introduced by the self-healing mechanism. We investigated three spare-cell placement strategies; UD, 0FA and MD. By using Monte Carlo Simulation we discovered that the best spare-cell placement strategy is MD, because the tabu search algorithm is allowed to move the spare-cells out of the way of the critical path, but provides a minimum distance to spares to be used in case of faults. Extensive experiments and a real-life example have shown the effectiveness of the proposed approach. Using the proposed setup, a designer can evaluate the right amount of redundancy to be introduced and can determine a mapping and placement such that the performance of the application is maximized, even in the case of permanent faults.

8. References

[1] M. R. Boesen, J. Madsen, "eDNA: A bio-inspired reconfigurable hardware cell architecture supporting self-organisation and self-healing", *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware Systems"*, pp. 147-154, 2009.

[2] J. Duato, S. Yalamanchili, N. Lionel, "Interconnection networks: An engineering approach", Morgan Kaufmann Publishers Inc, 2002.

[3] S. Bertozzi, A. Azquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems on chip: A feasibility study". *Proceedings of the Design, Automation and Test in Europe Conference,* DATE, 1, 2006.

[4] E. Carvalho, and F. Moraes, "Congestion-aware task mapping in heterogenous MPSoCs". *Internationally Symposium on System-on-Chip,* pp. 1-4, 2008.

[5] F. Glover and M. Laguna, "Tabu Search", Kluwer Academic Publishers, 19997.

[6] K. Chapman, "PicoBlaze 8-bit embedded microcontroller for Spartan-3, Virtex-II and Virtex-II Pro FPGAs", *Xilinx User Guide UG129 (v.1.1.2),* 2008.

[7] P. K. F., Holzenspies, G. J. M. Smit and J. Kuper, "Mapping Streaming applications on a reconfigurable MPSoC platform at run-time". *International Symposium on System-on-Chip,* pp. 1-4, 2007.

[8] ITRS. "Executive summary 2007 edition", *International Technology Roadmap for Semiconductors (ITRS)*, 2007.

[9] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems." *IEEE Transactions on Computers*, pp. 113-125, 2003.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: the mars approach", *IEEE Micro*, pp. 25-40, 1989.

[11] D. Mange, M. Sipper, A. Stauffer and G. Tempesti. "Toward robust integrated circuits: The embryonics approach", *Proceedings of the IEEE*, pp. 516-543, 2000.

[12] G. D. Micheli, "Synthesis and optimization of Digital Circuits", McGraw Hill Science, 1994.

[13] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications", *Proceedings Design, Automation and test in Europe, Conference and exhibition*, pp. 1164-1169,2004.

[14] T. Plaks, X. Zhang, G. Dragffy, A. Pipe, N. Gunton and Q. Zhu. "A reconfigurable self-healing embryonic cell architecture", *International Conference on Engineering of Reconfigurable Systems and Algorithms – ERSA '03*, pp. 134-140, 2003.

[15] S. Raychaudhuri. "Introduction to monte carlo simulation", *Proceedings – Winter Simulation Conference*, pp. 91-100, 2008.

[16] P. Pop, V. Izosimov, P. Eles, P. Zebo, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication", *IEEE Transactions on Very Large Scale Integation (VLSI) Systems",* pp. 389-402, 2009.

[17] M. R. Boesen, P. Schleuniger, J. Madsen, "Feasibility study of a self-healing hardware platform", *Proceedings of the 2010 Conference on Applied Reconfigurable Computing*, pp. 29-41, 2010.

[18]. M. R. Boesen, J. Madsen, D. Keymeulen, "Autonomous distributed self-organizing and self-healing hardware architecture – the eDNA concept", *Proceedings of the 2011 IEEE Aerospace Conference*, Big Sky, MT, 2011.

[19] M. R. Boesen, D. Keymeulen, J. Madsen, T. Lu, T. Chao, "Integration of the reconfigurable self-healing eDNA architecture in an embedded system", *Proceedings of the 2011 IEEE Aerospace Conference*, Big Sky, MT, 2011.

[20] A. Patooghy, S. G. Miremadi, "Complement routing: A methodology to design reliable routing algorithm for network on chips", *Microprocessor and Microsystems,* April 2010.