Experimental Allocation of Safety-Critical Applications on Reconfigurable Multi-Core Architecture

Louis Sutter*, Thanakorn Khamvilai*, Philippe Monmousseau*, John B. Mains*, Eric Feron*,
Philippe Baufreton†, François Neumann†, Madhava Krishna‡, S. K. Nandy‡,
Ranjani Narayan§ and Chandan Haldar§

*School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, USA
Email: {lsutter6, thanakornkhamvilai, philippe.monmousseau, jmains3, feron}@gatech.edu
†Safran Electronics & Defense, Massy, France
Email: {philippe.baufreton, francois.neumann}@safrangroup.com
‡CAD Laboratory, Indian Institute of Science, Bangalore, India
Email: madhav@cadl.iisc.ernet.in, nandy@cds.iisc.ac.in
§Morphing Machines Pvt. Ltd., Bangalore, India
Email: {ranjani, chandan}@morphing.in

***Abstract*—Multi-core processors pervade numerous industries but they still represent a challenge for the aerospace industry, where strong certification of every components is required. One way to make them enforce safety-criticality constraints is to ensure reallocation of critical tasks on the chip when they are affected by hardware faults.**

This paper describes and compares different models of a task reallocation problem for a reconfigurable multi-core architecture. It also presents the first version of the macroscopic model made of Raspberry Pi that was built to represent the multi-core architecture and to test the task allocation algorithm on an actual system, showing the increased robustness that the reallocation algorithm enables while cores are made faulty.

***Index Terms*—multi-core, reconfigurable, safety-critical, integer linear programming, Raspberry Pi**

I. Introduction

The world of embedded systems architecture is experiencing a major change with the onset of multi-core and many-core processors. They carry important benefits over single-core processors, bringing more computational power without augmenting chip's internal frequency, and thus without increased energy consumption or increased heating. Many industries are already taking advantage of such processors, and the aerospace industry could really benefit from the computational power and redundancy inherent to these processors.

Firstly, the computational power provided by parallelism would enable to execute computationally demanding applications on board, like engine health monitoring applications, which conventional embedded chips cannot process efficiently. This particular type of application has to be performed on board since airline companies cannot afford to stop a plane on the ground for a sufficient time to transfer the large data generated by engine sensors and analyze them out of the plane afterwards.

Secondly, the large number of available cores on a many-core processor enables graceful degradation on the chip: when some parts of the processor are subjected to hardware faults, affected applications can be reallocated to a different area of the processor and continue to support their intended functions. This process can ensure improved resilience of the system.

Despite these benefits, multi-core processors used in avionics systems represent a challenge for the aerospace industry. Since applications running in parallel on the chip share hardware resources, they may interfere with each other and safety-critical applications may be affected by non-critical ones, for example in the case they simultaneously try to access the same resources. This jeopardizes the determinism of software behavior running on multi-core processors and therefore represents an obstacle to the certification of their use in aircraft.

This paper presents several variations of a task allocation algorithm for a specific many-core architecture called REDEFINE[1], developed by the company Morphing Machines

This effort has been funded in part by SAFRAN and by the National Science Foundation, Grants CNS 1544332 and 1446758.

[1]REDEFINE is a registered trademark of Morphing Machines.

and the Indian Institute of Science (IISc), and applied to avionics control applications in collaboration with Safran Electronics & Defense. The implementations of the allocation algorithm differ from each other in the reallocation problem formulation and the type of solver they use. In one case, the solver is a commercial Integer Linear Programming (ILP) solver. In the other case, it is a Satisfiability (SAT) solver. Their performances are then compared for different problem sizes, that is the number of cores to handle on the processor.

We finally present a simple experiment that is the first step of the building of a macroscopic model of the REDEFINE architecture. This first experimental system shows how the allocation algorithm behaves on actual hardware and illustrates the increased robustness it enables. It also demonstrates the ability of the reallocation algorithm to detect simple hardware faults. Last, the hardware implementation enables the identification of more complex, actual faults.

II. REDEFINE Architecture

The REDEFINE many-core architecture [1] of interest to this paper is an architecture in development at the company Morphing Machines and the Indian Institute of Science. Unlike other architectures, REDEFINE features a dynamic reconfiguration capability: applications running on the chip can be dynamically allocated to different cores of the processor. This feature motivates the development of an algorithm that computes the allocation of applications according to the state of degradation of the chip and the criticality of each application.

We now present the main elements of the REDEFINE architecture to support the rest of the paper. For more details, the interested reader is invited to look at [2].

REDEFINE is made of two main elements: an Executable Fabric and a Resource Manager. It is connected to an external memory and a host, as shown in Fig. 1.

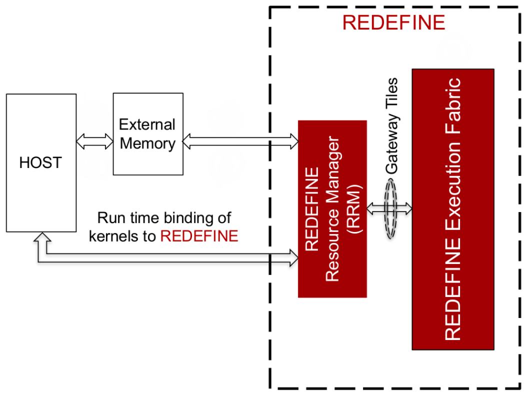

REDEFINE
HOST
External Memory
Run time binding of kernels to REDEFINE
REDEFINE Resource Manager (RRM)
Gateway Tiles
REDEFINE Execution Fabric

Fig. 1: The different components of the REDEFINE architecture.

The Execution Fabric is a toroidal mesh of a certain number of Tiles connected through the Network on Chip (NoC), as shown in Fig. 2. Each Tile includes a router (shown as a pink circle) and a Computer Resource (shown as a gray box and denoted CR thereafter) that is responsible for actual computations. Some extra Tiles are added on one edge of the Fabric to ensure the communication with the Resource Manager. These “Gateway Tiles” support routing functions only.

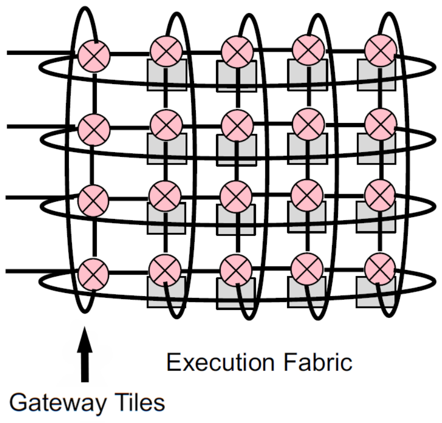


Fig. 2: Example of a toroidal mesh topology of the NoC for a 4×4 fabric. The pink circles represent routers, and the gray squares represent Compute Resources.

The REDEFINE Resource Manager (RRM) is the interface between the Fabric and the host that creates the decomposition of applications into sequences of basic elements called HyperOps. As such, the RRM is in charge of allocating parts of the Fabric to the different HyperOps, launching them, getting the results through the Gateway Tiles, and transferring them to the host.

The sequences of HyperOps are generated from C code during the offline compilation process, which also computes their spatial configuration that must be respected on the REDEFINE Fabric (Fig. 3). Each HyperOp is designed to be executed by one REDEFINE Tile, the basic element of the Fabric. The necessity to respect the orientation of the HyperOps pattern comes from the XY deterministic routing algorithm [3] used on the Fabric: to reach its destination, a packet first moves horizontally along the X-axis to reach the destination’s column and then vertically along the Y-axis to reach its row. HyperOps that need to communicate are therefore required to have a specific orientation.

An example of spatial configuration for an application computed by the compiler is given in Fig. 4. It gives the relative position of the HyperOps, also denoted “True Application Nodes” thereafter, that must be assigned to some Tiles of the Fabric. These True Application Nodes therefore require the CR of those Tiles. The router of some extra Tiles may also be

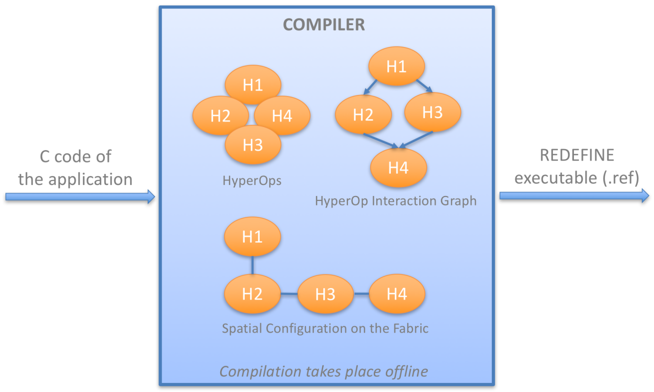

COMPILER
C code of the application
H1
H2
H4
H3
HyperOps
H1
H2
H3
H4
HyperOp Interaction Graph
REDEFINE executable (.ref)
H1
H2
H3
H4
Spatial Configuration on the Fabric
Compilation takes place offline

Fig. 3: Illustration of the compilation process.

needed for intra-application communication between Application Nodes. To ensure spatial partitioning of the applications on the Fabric and to prevent another application from using them, these extra Tiles are considered as entirely allocated to the application as well: the term “Ghost” Application Nodes then denotes fictitious Nodes that must be assigned to such Tiles.

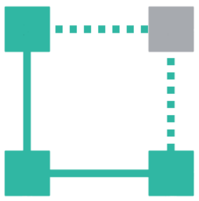


Fig. 4: Example of the spatial configuration of an application. Colored squares represent “True Nodes”, corresponding to a Tile whose CR must be allocated to the application. A “Ghost” Application Node (in gray) is added because the top right Tile’s router is used by the application for intra-application communication. Such a Node is considered to be part of the application.

Because of its supervisory role, the RRM is in charge of executing the task allocation algorithm detailed in this paper. This implies that the RRM is aware of the faults occurring on the Fabric to compute an appropriate allocation. A simple model of faults is considered, in which each component of a Tile, the CR and the router, can be either functional or permanently faulty.

To detect these faults, an appropriate observation system is used. This observation system does not necessarily require instrumentation of the hardware since some checks can be performed at the software level. For example, sending a message to the different Tiles to check that they respond, as is shown in the experiment below (see section V), enables the detection of router faults, whereas a triple redundant system implementation can identify computational errors corresponding to a CR fault. Detecting faults at a lower level thanks to an instrumentation can also be imagined, for example by checking voltage of some specific points in the architecture. The goal of the macroscopic model presented below is also to experiment with other ways to detect faults.

III. New task allocation algorithm

A. Motivations

Reference [2] introduces a first task allocation algorithm for the REDEFINE architecture that is able to compute a new allocation of the applications on the NoC, given the following information: the size of the REDEFINE Fabric, the status of the Tiles, and a set of applications with their pattern as computed by the compiler. This algorithm is based on an Integer Linear Programming (ILP) formulation of the allocation problem that is solved using the Gurobi [4] commercial ILP solver each time a fault is detected.

After having studied this first algorithm and its implementation, it appears that the solver and the optimization problem formulation can be improved.

1) Modification of the solver: It turns out that the problem was formulated as a *Pseudo-Boolean problem*, that is, an ILP problem with binary variables only [5]. The constraints are Pseudo-Boolean (PB) constraints i.e. inequalities on a linear combination of boolean variables:

$$A_1x_1 + A_2x_2 + ... + A_nx_n \geq K$$

where $A_1, A_2, ..., A_n, K$ are real constants and $x_1, x_2, ..., x_n \in \{0, 1\}$.

Such Pseudo-Boolean problems can be converted into Satisfiability (SAT) problems [6] and use efficient open-source SAT solvers like MiniSat, detailed in [7]. Since MiniSat is written in C++ and includes an extension, MiniSat+ detailed in [6], to handle PB problems, it offers a low-cost and free alternative to Gurobi, having in mind an implementation on the REDEFINE architecture.

2) Modification of the problem formulation: The problem formulation in [2] presents two characteristics that can be improved.

First, the solver may be called up to three times for one fault. This occurs when the first try, reallocating only the affected application, and the second try, reallocating the affected application and the lower-priority ones, both lead to an unfeasible problem. The third call is then performed after having dropped all lower-priority applications (see [2] for more details).

Second, the optimization problem formulation given in [2] drops all non-critical applications when a fault affects a safety-critical application that cannot be relocated without touching the other applications.

The alternate optimization problem formulation given below allows the allocator to be less conservative in terms of the number of applications being dropped. It also requires at most one SAT solver call per failure, while allowing more applications to run.

B. Definitions and parameters

Before giving the new formulation of the problem, some definitions and parameters concerning the mathematical representation of the architecture must be given.

The Network on Chip's topology is described by an undirected graph. Each vertex represents a Tile and each edge represents a communication link between the routers of two Tiles. Such a link is called a "NoC Path".

The matrix G is the $N_T \times N_{paths}$ incidence matrix of the graph representing the NoC, where N_T is the number of Tiles on the fabric and N_{paths} is the number of NoC Path.

$$G_{ij} = \begin{cases} 1 & \text{if the Tile } i \text{ and the NoC Path } j \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

The N_{apps} different applications that are to be executed on the platform are represented by N_{apps} undirected graphs. On REDEFINE, the compiler computes the number of Compute Resources (and their spatial configuration) that each application requires, with the objective of maximizing parallelism. Therefore, each vertex of a graph represents a Compute Resource that will be assigned to the application and each edge represents a communication path that links the routers associated to two of these Compute Resources.

The matrix A^k is the $N^k_{nodes} \times N^k_{links}$ incidence matrix representing the graph of application k, where N^k_{nodes} is its number of nodes, denoted "Application Nodes" (corresponding to the HyperOps mentioned in section II), and N^k_{links} is its number of edges, denoted "Application Links".

$$A^k_{ij} = \begin{cases} 1 & \text{if the Node } i \text{ and the Link } j \text{ of application} \\ & \text{graph } k \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

$N_{nodes} = \sum_{k=1}^{N_{apps}} N^k_{nodes}$ is the total number of Application Nodes.

$N_{links} = \sum_{k=1}^{N_{apps}} N^k_{links}$ is the total number of Application Links.

An overall application graph represented by the incidence matrix A is constructed from the A^k $(k = 1, \ldots, N_{apps})$ matrices as such:

$$A = \begin{bmatrix} A^1 & & \\ & \ddots & \\ & & A^k \end{bmatrix} \quad \text{is a } N_{nodes} \times N_{links} \text{ matrix.}$$

Applications Nodes and Links receive therefore a global numbering corresponding to their position in this overall matrix.

C. New formulation

In the new formulation presented here, allocating or dropping an application is captured by additional boolean variables, such that one call to the SAT solver is sufficient to determine the optimal choice of applications to reallocate and to drop. Applications not directly affected by the fault are allowed to move, though penalties are introduced to prevent unnecessary moves.

1) General form of the formulation: As explained previously, the allocation problem is formulated as a Pseudo-Boolean problem of the form:

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & M_1\mathbf{x} \le \mathbf{b_1} \\ & M_2\mathbf{x} = \mathbf{b_2} \\ \text{and} & \mathbf{x} \text{ is a binary vector.} \end{array}$$

$\mathbf{x}$ is the vector of decision variables, $\mathbf{c}$ the coefficients of the objective function and M_1, M_2, $\mathbf{b_1}$ and $\mathbf{b_2}$ are parameters that derive from the aggregation of the constraints of the problem.

2) Decision variables: The decision variables of the problem describe the mapping of the Application Nodes to the Compute Resources of the Tiles on the NoC and the mapping of the Application Links to the NoC communication paths. In addition, two types of variables are used to represent the decision of dropping or moving an application. The vector $\mathbf{x}$ is thus made up of:

$$X^{CR\to apps}_{ij} = \begin{cases} 1 & \text{if the CR of Tile } i \text{ is allocated} \\ & \text{to the Application Node } j \\ 0 & \text{otherwise} \end{cases}$$

$$X^{paths\to links}_{ij} = \begin{cases} 1 & \text{if the NoC Path } i \text{ is allocated to} \\ & \text{the Application Link } j \\ 0 & \text{otherwise} \end{cases}$$

$$r_k = \begin{cases} 1 & \text{if the application } k \text{ is running} \\ 0 & \text{if it is dropped} \end{cases}$$

$$M^N_j = \begin{cases} 1 & \text{if the Application Node } j \text{ is} \\ & \text{moved from its previously allo-} \\ & \text{cated CR} \\ 0 & \text{otherwise} \end{cases}$$

where $X^{CR\rightarrow apps}$ is a $N_T \times N_{nodes}$ matrix; $X^{paths\rightarrow links}$ is a $N_{paths} \times N_{links}$ matrix; r is a $N_{apps} \times 1$ matrix and M_j is a $N_{nodes} \times 1$ matrix.

3) Objective function: The objective function is used in order to maximize the number of executed applications while minimizing the number of reallocation:

$$\max \left(\sum_{k=1}^{N_{app}} r_k - \sum_{j=1}^{N_{nodes}} \frac{M_j^N}{N_{nodes}+1} \right) \tag{1}$$

where r_k and M_j^N are the decision variables previously introduced.

The coefficient $\frac{1}{N_{nodes}+1}$ ensures that an application is always moved rather than dropped.

4) Constraints:

a) Binary variables: All the decision variables are binary.

b) Resource allocation and partitioning: Several equations express the constraints of allocating the resources of the Fabric to applications while enforcing partitioning on the chip.

First, a CR can be allocated to at most one application, as a way to enforce spatial partitioning of applications on the NoC, i.e.

$$\forall i = 1, \ldots, N_T, \ \sum_{k=1}^{N_{nodes}} X_{ik}^{CR\rightarrow apps} \leq 1. \tag{2}$$

Each running Application Node must be assigned to exactly one CR, i.e.

$$\forall i = 1, \ldots, N_{nodes}, \ \sum_{k=1}^{N_T} X_{ki}^{CR\rightarrow apps} = r_{k_N(i)}. \tag{3}$$

$k_N(i)$ is the application number corresponding to Application Node i.

A NoC Path can be allocated to at most one Application Link[2], i.e.

$$\forall i = 1, \ldots, N_{paths}, \ \sum_{k=1}^{N_{links}} X_{ik}^{paths\rightarrow links} \leq 1. \tag{4}$$

[2]This does not mean that this NoC Path cannot be used for other communication purposes on the NoC, but only one of the Application Link computed by the compiler for the applications can be allocated to that NoC Path.

Each running Application Link must be assigned to exactly one NoC Path, i.e.

$$\forall i = 1, \ldots, N_{links}, \ \sum_{k=1}^{N_{paths}} X_{ki}^{paths\rightarrow links} = r_{k_L(i)}. \tag{5}$$

$k_L(i)$ is the application number corresponding to Application Link i.

c) Conformity to the architecture: An Application Link that connects two Application Nodes must be allocated to a NoC Path connecting the two CRs on which those two nodes have been mapped, i.e.

$$X^{CR\rightarrow apps} \ A = G \ X^{paths\rightarrow links}. \tag{6}$$

d) Reallocating several applications: A given Application Node can either remain affected to the same Tile, either be moved, either be dropped:

$$\begin{gathered} \forall i = 1, \ldots, N_T, \ \forall j = 1, \ldots, N_{nodes}, \\ (1 - r_{k_N(j)}) + M_j^N + X_{ij}^{CR\rightarrow apps} = X_{old\ ij}^{CR\rightarrow apps}, \end{gathered} \tag{7}$$

with $X_{old}^{CR\rightarrow apps}$ the parameter containing the mapping between CR and Application Nodes computed during the previous allocation.

This constraint is ignored for the initial allocation.

e) Priorities: The choice is made to execute the safety-critical application for all time, and then to execute applications with low priority only if every applications with higher priority can be executed. This choice ensures that the allocation algorithm does not drop a high-level-priority application to execute several low-level-priority ones that requires fewer CR. This is written:

$$\begin{gathered} r_1 \geq 1 \\ \forall k = 2, \ldots, N_{app}, \\ r_{k-1} \geq r_k. \end{gathered} \tag{8}$$

Application 1 has the highest priority and application N_{app} has the lowest one.

f) Faults: Constraints inherited from [2] make the algorithm take into account faulty CRs or routers. Within a Tile i:

- If both the Compute Resource and the router are healthy, any Application Node can be mapped on the tile.
- If the Compute Resource is faulty but the router is healthy, only "Ghost" Application Nodes can be mapped on the Tile i.

$$\sum_{k \in true\ nodes} X_{ik}^{CR\rightarrow apps} = 0 \tag{9}$$

- If the router is faulty, regardless of the health of the Compute Resource, then no Application Nodes can be mapped on the Tile i.

$$\sum_{k=1}^{N_{nodes}} X_{ik}^{CR \rightarrow apps} = 0 \tag{10}$$

Hence, a reallocation is needed each time a CR fault occurs on a Tile where a "True" Node was mapped or a router fault occurs on a Tile where any node was mapped. The detection of these fault is assumed for the model but needs to be implemented in practice.

g) Spatial Orientation: To ensure correct orientation of applications, a set of constraint used in [2] is also needed. In order to enforce this, the numbering of the Tiles on the NoC is used. For example, as illustrated in Fig. 5, a Tile has always a number difference of -1 with its right neighbor and $+N_{/row}$ with its top neighbor, where $N_{/row}$ is the number of Tile per row of the NoC ($N_{/row} = 4$ in our example).

The difference between the numbers of the contiguous pairs of Tiles allocated to an application must match the orientation computed by the compiler.

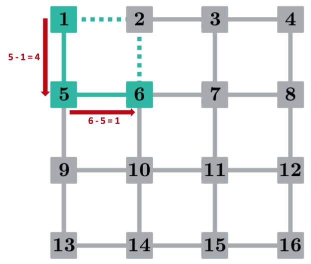


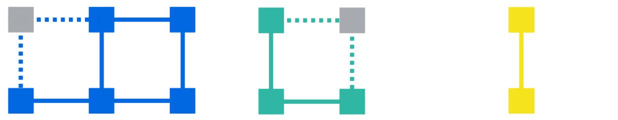
Fig. 5: By ensuring that the difference between two Tiles' numbers allocated to an application is equal to a specific number, the spatial orientation of the application can be enforced.

IV. Algorithms comparison

The previous modifications enabled to obtain four different allocation algorithms by choosing one the two available solvers, CVX/Gurobi or MiniSat+, and one model of the allocation problem, either the first formulation detailed in [2] or the new one presented here. In examples, the three applications shown in Fig. 6 are considered.

After having forced the same initial configuration for the two algorithms, their behavior are compared for the same fault sequence in Fig. 7.

The second algorithm enables to better use the remaining Tiles of the REDEFINE Fabric so that more applications can continue to be executed, even after a high-priority application is affected.



(a) Application 1, highest priority, critical (b) Application 2, intermediate priority, non-critical (c) Application 3, lowest priority, non-critical

Fig. 6: The spatial configurations of the three different applications that are to be executed on the platform. The gray squares represent the "Ghost" Nodes of the applications, which only use the router of the Tile for intra-application communication.

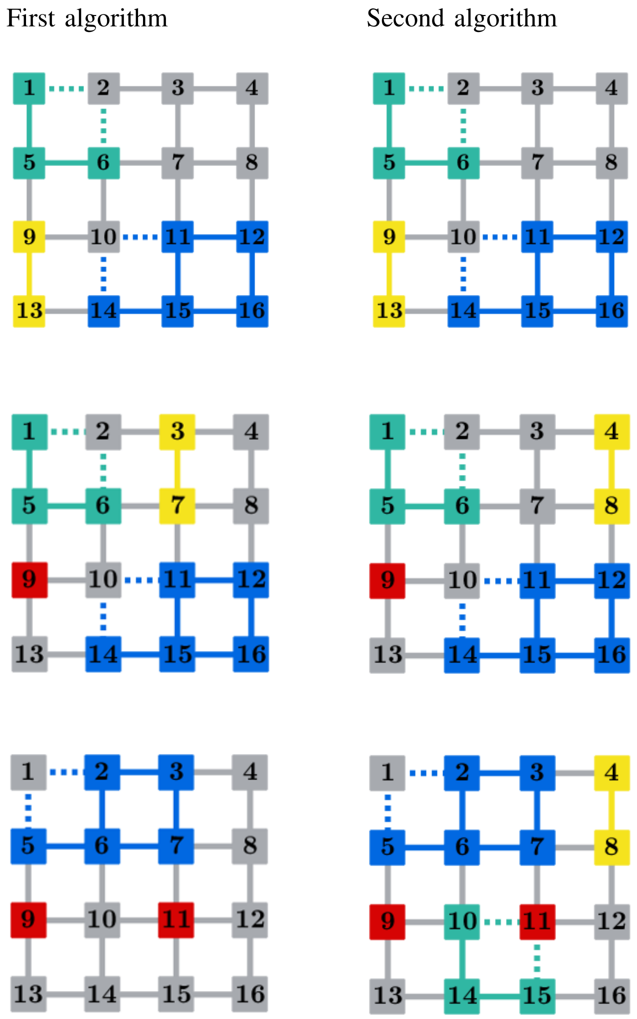


Fig. 7: Comparison of the two algorithms for the same CR faults sequence, represented by red squares. When the fault affects the critical application, the first algorithm drops all the other ones, contrary to the second one who manages to keep all the applications.

This algorithm can require an execution time per call from 20% to 60% higher than the previous algorithm. However, its advantage is that it is called no more than once after each failure, whereas the previous one could be called up to three times, which is profitable on average as seen below.

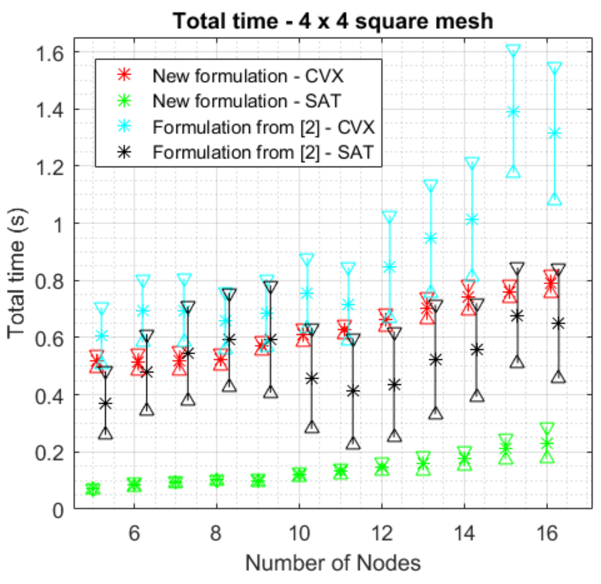
Considering the different choices of couple Algorithm - Solver, their relative performance were evaluated to choose the one that is the most appropriate for the macroscopic model being built.

To compare their performance for a given network size, for example a 4×4 square mesh, random sets of applications and random sequences of faults were generated, before measuring the required time for the couple Formulation - Solver to compute the new allocation. For this test, the algorithms were all run using Matlab, combined with the CVX modeling framework [8] and the solver Gurobi [4] on the one hand, and on the other hand combined with the SAT-solver MiniSat [7] [9] and its extension for PB problems MiniSat+ [6] [9], both written in C++ and called from Matlab.

The results of this comparative study are given in Fig. 8 for both a 4×4 and a 6×6 square mesh. The raw results were the computation time after each failure depending on the number of Application Nodes to handle, which is the main parameter determining the size of the problem to solve. Then, the mean time as well as the standard deviation were plotted for every Node number, in order to quantify more precisely how consistent each of the four couples Algorithm - Solver is.

On a relatively small network of 4×4 Tiles, the comparison in Fig. 8 shows that the new formulation, corresponding to red and green plots, is more regular in terms of computation time since one call to the solver per failure is ensured in this new version. In addition, the SAT solver enables to obtain smaller mean times for small numbers of applications nodes. Therefore, the couple New Formulation - MiniSat appears as the best choice for the 4×4 network used for the experiment.

However, as the size of the network increases, the mean computation time for this algorithm grows quickly and with high standard deviation. It becomes then necessary to use the new formulation with the CVX / Gurobi solver that benefits from a better scalability.



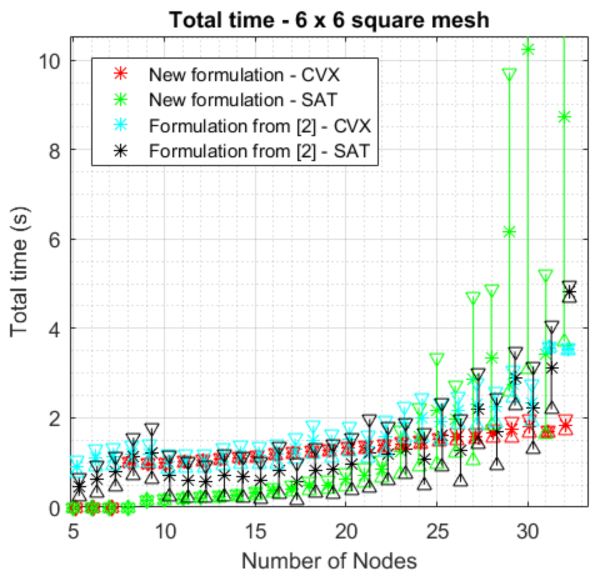


Fig. 8: Statistical comparison of the computation time of the different algorithms for random applications and faults on a 4×4 and 6×6 mesh, as a function of the number of Application Nodes to handle. Stars $*$ represent the mean time and the distance between two triangles Δ represent the standard deviation. Times were obtained on a personal laptop with an Intel Core i7 CPU at 2.40 GHz and 8 GB of RAM.

V. Demonstration

A. *Hardware Replication*

As previously discussed in section II, the REDEFINE chip is still being developed; however, its multi-core architecture can be reproduced by using a group of small embedded computers. Each of these computers is called a "Tile", and represents one core of the actual REDEFINE architecture. The subtask of software running on one of these Tiles is called an "Application Node".

To replicate a 4×4 chip fabric, 17 Raspberry Pi computers are used: the first 16 ones are to represent the Tiles of the Fabric, and the last one acts as the resource manager (RRM) as shown in Fig. 10. The Tile runs the user-defined Application Node and the fault actuating and detection system, which updates its own status to the resource manager. The resource manager computes allocations based on tiles' status, by using the algorithm discussed in section III, and transmits back the output.

The Tiles that fail to update their own status within a timeout limit are considered as “faulty Tiles”. The non-faulty Tiles that are not running any application nodes at a given configuration are called “free Tiles”.

B. Fault Actuating and Detection System

Two types of high-level hardware faults, which are faulty compute resources and faulty routers, were addressed in [2]. To reproduce these faults, the fault actuating system is designed using two ON/OFF switches (Fig. 9) corresponding to each type of failures in order to allow a demonstrator to decide which faults to occur on which Tiles. Once the switch is triggered, the voltage is captured, converted to a digital data, and forwarded to the resource manager as the current status of the Tile.

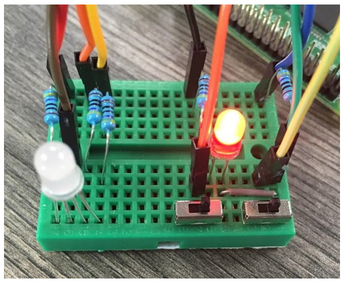


Fig. 9: Hardware associated with each Raspberry Pi Tile. The RGB-LED (bottom-left corner) represents the LED application. The red LED (right side) indicates an healthy Tile when turned on. Two switches are for triggering two types of faults.

C. Scenario

In the demonstration, three applications with a distinct usage of computer resources and a unique spatial configuration (Fig. 10) are considered and represented by a specific color of RGB-LED, allowing us to visualize the reconfiguration of Application Nodes. Each of these applications also has the relative priority based on its safety-criticality for reconfiguration, meaning that the algorithm is allowed to shut down the lower-priority applications to maintain the execution of the higher-priority ones in case of Compute Resources shortage. The LED color assigned to each application, its relative priority, and its spatial configuration are given by Table I.

TABLE I: Three applications’ LED colors, relative priorities, and spatial configurations.

Application	Color	Priority	Spatial Configuration
1^{st}	Blue	Highest	2-row by 3-column
2^{nd}	Green	Intermediate	2-row by 2-column
3^{rd}	Yellow	Lowest	2-row by 1-column

Remark 1: The free Tiles are represented by turning on the RGB-LED to the white color, and the faulty Tiles are represented by turning both LEDs off.

Remark 2: The red LED is controlled solely by the analog input voltage from the switches. Since there is no software associated with this LED, it properly indicates a faulty Tile when turned off.

D. Communication Protocol between Tiles

To establish the communication between Raspberry Pi Tiles, a local area network (LAN) has been developed. In this network, every Tile is connected to a common routing switch and communicates through a MQTT protocol [10]. This protocol is a publish-subscribe-based messaging network protocol working as a surface layer of the standard TCP/IP protocol. Every data published or subscribed using MQTT has to be associated with its own unique “Topic”, and is queued by the central handler called “Broker”

To implement the MQTT protocol for this demonstration, the resource manager runs the Open-source MQTT-broker named Mosquitto [11], subscribes to a status topic from every single Tile, and publishes the result of the allocation algorithm. On the other hand, each Tile only subscribes to the result topic of the resource manager, and publishes its own status.

Note that the XY routing algorithm mentioned in section II can be implemented as a constraint on the software-level of the communication protocol; however, this is not required on the LED application because there is no communication occurring between the Application Nodes.

E. Results

The demonstration result is shown by a sequence of pictures as in Fig. 11. This demonstration is initialized as shown in Fig. 10. Then, the demonstrator randomly makes the Tiles become faulty by either switching off, unplugging the power cable, or removing the Ethernet cable. Fig. 11a, 11c, 11e, and 11g show which Tiles are faulty while Fig 11b, 11d, 11f, and 11h show the new configuration according to the current faulty Tiles.

Note that the faulty Tile that came from removing the Ethernet cable still had its RGB-LED lights up, but it lost the connection to the entire system. Therefore, the resource manager considered it as a faulty Tile.

VI. Conclusion

The task allocation algorithm presented in our last paper has been improved at two levels. Firstly, a new formulation of the task allocation problem enables to add more flexibility

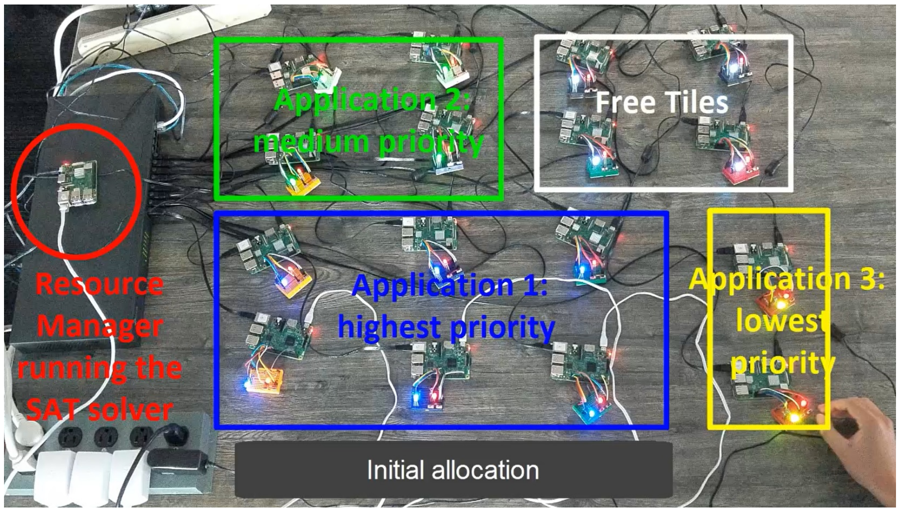

Application 2: medium priority
Free Tiles
Resource Manager running the SAT solver
Application 1: highest priority
Application 3: lowest priority
Initial allocation

Fig. 10: Hardware setup using Raspberry Pi computers for REDEFINE Tiles and the resource manager.

in the algorithm by allowing the reallocation of several applications, and therefore enables to keep more applications running on the architecture. Moreover, a unique call to the solver for each reallocation is now guaranteed. Secondly, instead of the commercial ILP solver used previously, a SAT solver has been used to solve the ILP formulation of the reallocation problem. This second solver has the benefits to be open-source and quite easily embeddable.

The different versions of the task allocation algorithm have been compared in terms of behavior and computation time. For relatively small architecture of about twenty cores, the new formulation coupled with the SAT solver is the most efficient. However, tests on bigger architectures show that the scalability is better for the commercial solver coupled with the new formulation.

This paper also presented the experiment that was built. The experiment shows the execution of the reallocation algorithm on concrete hardware while enabling to implement some basic fault detection systems that are required for the algorithm. This experiment is also the first step in the building of a macroscopic model of the REDEFINE architecture hosting the reallocation algorithm.

The next steps for this macroscopic model include implementing complementary fault detection systems, for example ones enabling to detect computation faults. One possible solution is the use of Triple Redundancy of the applications on the multi-core Fabric, which would also have the benefit of ensuring continuity of service of the application during the reallocation of the faulty unit, since the two healthy units can continue to run. An other important step will be to run non trivial parallel applications on the Fabric, like control applications.

Finally, this macroscopic model of REDEFINE will help to improve the actual REDEFINE architecture in terms of safety for a potential use in avionics systems, for example, by exploring ways of removing the single-point of failure that represents the Resource Manager by distributing its tasks on the Fabric itself, which may lead to some interesting questions.

References

[1] M. Alle, K. Varadarajan, A. Fell, C. R. Reddy, J. Nimmy, S. Das, P. Biswas, J. Chetia, A. Rao, S. K. Nandy, and R. Narayan, "REDEFINE: runtime reconfigurable polymorphic ASIC," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 2, 2009.

[2] T. Guillaumet, E. Feron, P. Baufreton, F. Neumann, K. Madhu, M. Krishna, S. K. Nandy, R. Narayan, and C. Haldar, "Task allocation of safety-critical applications on reconfigurable multi-core architectures," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, Sept 2017, pp. 1–10.

[3] V. Rantala, T. Lehtonen, J. Plosila *et al.*, *Network on chip routing algorithms*. Turku Centre for Computer Science, 2006.

[4] "Gurobi optimizer reference manual," *Gurobi Optimization, Inc.*, 2016, http://www.gurobi.com.

[5] Dingzhu Du, Jun Gu, Panos M. Pardalos, *Satisfiability Problem: Theory and Applications*. American Mathematical Society, 1997.

[6] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," in *Journal on Satisfiability, Boolean Modeling and Computation 2*, 2006, pp. 1–25.

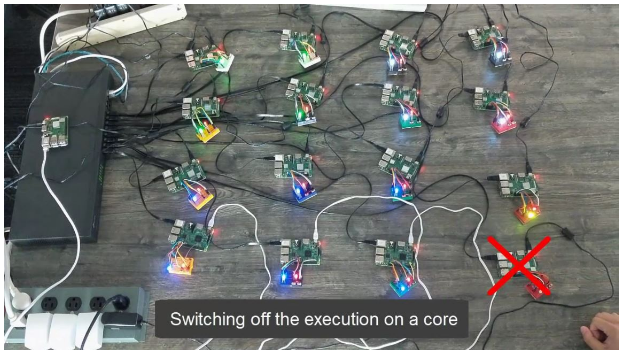
[7] ——, "An Extensible SAT-solver [extended version 1.2]," in *6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[8] M. Grant and S. Boyd, "CVX: Matlab Software for Disciplined Convex Programming, version 2.1," http://cvxr.com/cvx, March 2014.

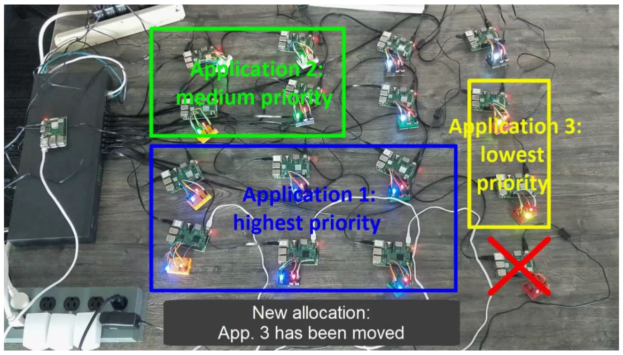
[9] N. Eén and N. Sörensson, "The MiniSat Page," http://minisat.se/.

[10] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S x2014; A publish/subscribe protocol for Wireless Sensor Networks," in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, Jan 2008, pp. 791–798.

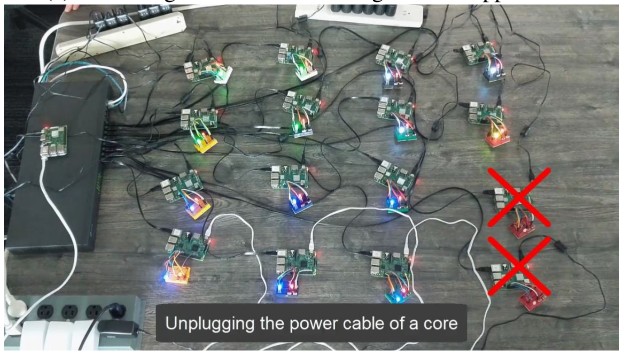
[11] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *The Journal of Open Source Software*, vol. 2, no. 13, p. 265, may 2017. [Online]. Available: https://doi.org/10.21105/joss.00265



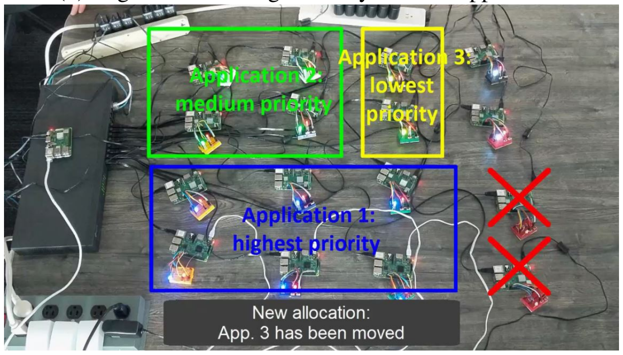
(a) Switching off one Tile running the 3rd application



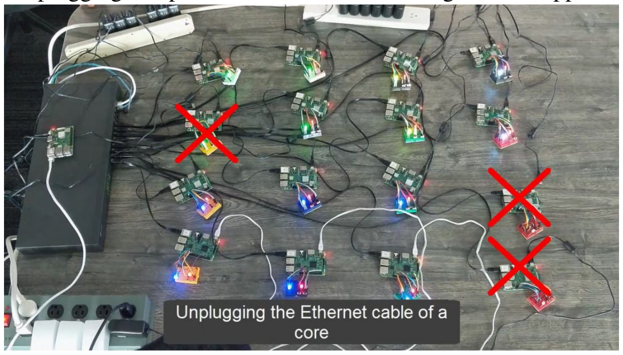
(b) Algorithm reconfigures only the 3rd application



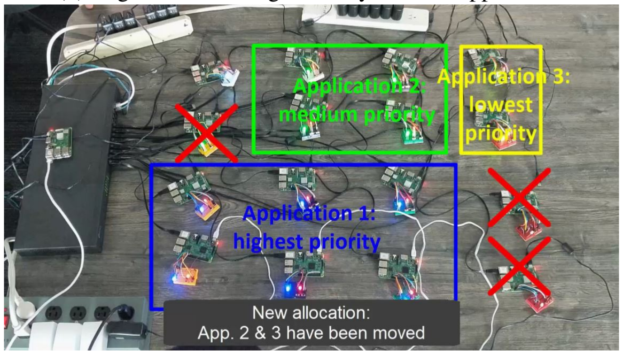
(c) Unplugging the power from one Tile running the 3rd application



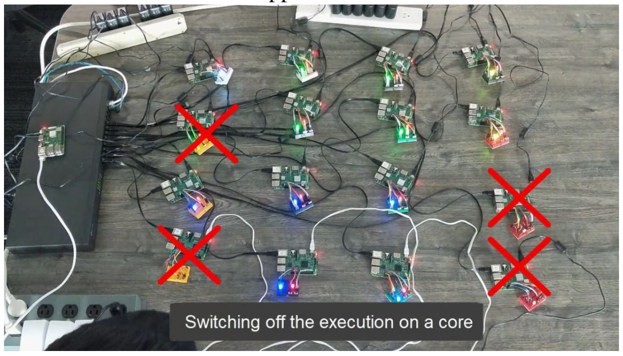
(d) Algorithm reconfigures only the 3rd application



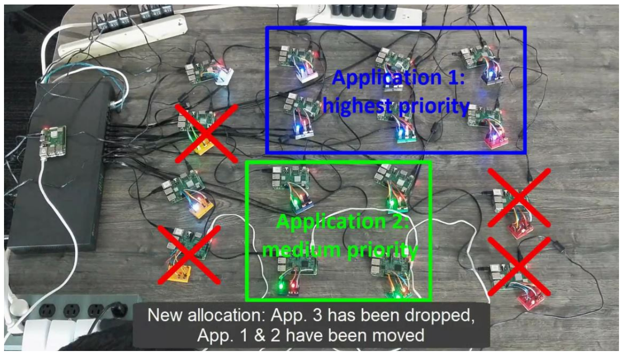
(e) Removing the Ethernet cable from one Tile running the 2nd application



(f) Algorithm reconfigures the 2nd and the 3rd application



(g) Switching off one Tile running the 1st application


Application 1:
highest priority
Application 2:
medium priority
New allocation: App. 3 has been dropped,
App. 1 & 2 have been moved

(h) Algorithm drops the 3rd application and reconfigures the 2nd and the 1st application

Fig. 11: Result of the task allocation algorithm. A full video of the demo is available on the clickable link https://www.dropbox.com/s/5xbz2d13xjz4zmw/demo_video_07_18.mp4?dl=0