Task Allocation of Safety-Critical Applications on Reconfigurable Multi-Core Architectures with an Application on Control of Propulsion System

Thanakorn Khamvilai*, Louis Sutter*, 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: {thanakornkhamvilai, lsutter6, jmains3}@gatech.edu,
eric.feron@aerospace.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*—In the aerospace industry, the highest criticality standard is required for the certification of avionics systems. A multi-core processor with reconfiguration capabilities where safety-critical applications are reallocated once they are affected by faults is one efficient way to enforce such criticality constraints.**

This paper presents a new model of a task reallocation problem for a reconfigurable multi-core architecture, which allows an execution of lower priority applications when the resources for executing the higher application are insufficient. Furthermore, it provides an implementation of an actual cyber-physical system: the control of a propulsive system with three redundant controllers. In addition to the fault injection mechanisms, a fault recovery capability and a fault detection system based on a majority rule voter are included.

***Index Terms*—multi-core, reconfigurable, safety-critical, integer linear programming, propulsion system, fault tolerance, voter**

I. Introduction

With the onset of multi-core and many-core processors, the world of embedded systems is experiencing a significant adaptation [1]. By transitioning to multi-core architecture, many advantages have been carried over single-core processors. For example, multi-core systems can provide more computational power without augmenting chip's internal frequency; thus, there is no added energy consumption or heat. Some major industries are already taking advantage of such processors, such as the automotive industry [2], the biotechnology industry [3] and the circuit industry [4]. In spite of these benefits, one aspect of multi-core processors remains challenging for the aerospace industry. Since hardware resources are shared, interference is possible and safety-critical applications may be affected by non-critical ones, for instance when 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 the aircraft that would use them without proper adaptation. Such an obstacle can however be overcome with appropriate partitioning of the tasks and a reallocation process using the inherent redundancy of computing elements on a multi-core processor.

This paper presents a new formulation of a task allocation algorithm for a multi-core architecture. Although the algorithm could be used in general, it is designed specially for the one called REDEFINE[1], developed by the company Morphing Machines and the Indian Institute of Science (IISc), and applied to avionics control applications in collaboration with Safran Electronics & Defense.

Besides the new reallocation algorithm, the contribution of this paper also includes an experiment that is the building of a macroscopic model of the REDEFINE architecture. This experiment on embedded systems shows how the reallocation algorithm behaves on an actual cyber-physical application of controlling the propulsion system. Furthermore, it uses a majority rule vote as a building block to detect hardware faults.

The rest of this paper is organized as follows. First, the architecture of REDEFINE is restated in Section II. Then,

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.

978-1-7281-0649-6/19/$31.00 ©2019 IEEE

the new formulation of our reallocation algorithm is provided in Section III. The comparison of this formulation with the one in [5] is given in Section IV. A new experimental setup, including a propulsion system, is presented in Section V and the experimental results are discussed in Section VI. Mathematical proofs of the successful reallocation ensured by the algorithm are given in an Appendix.

II. REDEFINE Architecture

The REDEFINE many-core architecture has been previously described in details in [6] and [7]. We will only state its physical architecture to support the rest of the paper.

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.

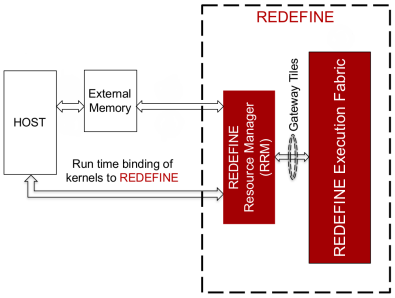


Figure 1: The components of the REDEFINE architecture.

The Executable Fabric is a topology mesh of a certain number of Tiles connected through a Network on Chip (NoC), as shown in Fig. 2. Each Tile includes a router (symbolized by a pink circle) and a Computer Resource (symbolized by a gray square 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.

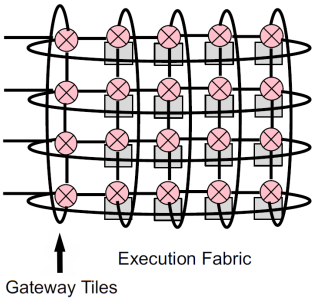


Figure 2: Example of a toroidal mesh topology of the NoC.

The REDEFINE Resource Manager (RRM) is the interface between the Fabric and the host that creates the decomposition of applications, given in Fig. 4, into sequences of basic elements, generated from C code during the offline compilation process, 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.

III. A new task allocation system

A. Motivations

The authors' previous works [7] and [5] proposed task allocation algorithms based on an Integer Linear Programming (ILP) formulation which is iteratively solved on-line by a mathematical program solver each time a fault occurs on a computer resource. The solver can compute a new allocation of the applications on the NoC, given the following information:

- The structure of the REDEFINE fabric
- The health status of the CRs
- The set of applications with their configuration

The solver will maximize the number of applications running on the system, with a penalty for reallocations and a reward for applications with high priority. At this stage, the solver does not take into account constraints to dedicate a communication channel through the NoC to each application in order to communicate with the host.

Even though the algorithm presented in [5] is the improvement of the one in [7], its solver as well as its optimization problem can be further enhanced.

1) New solver: The solver used in [7] is GUROBI [8], a commercial software operated under academic license. In [5], "MiniSat" solver [9] had been chosen because the mixed-integer program that represents the allocation problem enables its formulation as a Satisfiability problem (SAT) [10]. However, this problem translation adds an extra step to the solver before solving the problem. Hence, this additional procedure reduces the performance, in term of its execution time, of the NoC as the number of CRs increases [5].

The alternative solver used in this paper is "GNU Linear Programming Kit" (GLPK) [11], which is a light-weight, open-source software written in ANSI C and can be installed on on the NoC emulator described in [7] and thereafter. Thus, this solver gives a benefit on the scalability over the two previously mentioned solvers.

2) New problem formulation: The aspect of the problem formulation in [5] that can be improved is described as follows.

Let's consider an application with intermediate priority that requires a large number of CRs to run. When faults occur on the Fabric, low-priority applications will be dropped first. After several faults, the considered intermediate-priority application will be dropped as well to maintain the execution of high-priority applications. However, by dropping this "large" intermediate-priority application, the CRs it was using become free again. The motivation of this new formulation

is to allow the execution of "small" low-priority applications on those new free CRs, even though the intermediate-priority application is not running anymore. The formulation in [5] did not allow this but, instead, enforced a strict priority order for the running applications; *i.e.* an application can run only if **all** applications with higher priority are executed. Yet, it must be guaranteed that this behavior happens only if there is no more space for applications with high or intermediate priority, to prevent the algorithm from favoring many "small" low-priority applications rather than one "large" high-priority or safety-critical application.

The formulation is provided as follows, along with the proof that high-priority applications still continue to be executed if there are enough healthy CRs.

B. Definitions and parameters

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

1) Definitions of the System's Topology:

Definition 1. The Network on Chip's topology consists of *Computer Resources*, each pair connected by communication links called *NoC Paths*. Then, N_{CRs} is defined as the number of Computer Resources and N_{paths} is defined as the number of NoC Paths.

Definition 2. Each application will be designated by its index. N_{apps} is defined as the total number of applications that we wish to execute. Applications are ranked in priority order, with application 1 having the highest priority, and application N_{apps} having the lowest priority.

Definition 3. For $k \in [\![1, N_{\mathrm{apps}}]\!]$, application k consists of *Application Nodes*, each pair connected by communication links called *Application Links*. Then N^k_{nodes} is defined as the number of Application Nodes and N^k_{links} is defined as the number of Application Links in application k.

Definition 4. $N_{\mathrm{nodes}} \coloneqq \sum_{k=1}^{N_{\mathrm{apps}}} N^k_{\mathrm{nodes}}$ is the total number of Application Nodes, and $N_{\mathrm{links}} \coloneqq \sum_{k=1}^{N_{\mathrm{apps}}} N^k_{\mathrm{links}}$ is the total number of Application Links.

Definition 5. Let $\mathcal{G} = (V, E)$ be a graph where V represents a set of vertices, and E represents a set of edges. A graph $\mathcal{G}$ is said to be *a graph representation* of the NoC *iff* the vertex $v_i \in V$ solely represents the i-th CRs, and the edge $v_iv_j \in E$ connecting v_i and v_j solely represents the NoC Path that connects the i-th CR and the j-th CR. Similarly, a graph $\mathcal{G}' = (V', E')$ is said to be *a graph representation* of an application *iff* the vertex $v'_i \in V'$ solely represents the i-th Application Node, and the edge $v'_iv'_j \in E'$ connecting v'_i and v'_j solely represents the Application Link that connects the i-th Application Node and the j-th Application Node.

Definition 6. Given a NoC graph representation $\mathcal{G} = (V, E)$, the $N_{\mathrm{CRs}} \times N_{\mathrm{paths}}$ *NoC unoriented incidence matrix* G associated with $\mathcal{G}$ is defined as:

$$[G]_{ij} = \begin{cases} 1 & \text{if } v_i \in V \text{ and } e_j \in E \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

Definition 7. Given the k-th application graph representation $\mathcal{G}^k = (V^k, E^k)$, the $N^k_{\mathrm{nodes}} \times N^k_{\mathrm{links}}$ *application unoriented incidence matrix* A^k associated with $\mathcal{G}^k$ is defined as:

$$[A]^k_{ij} = \begin{cases} 1 & \text{if } v^k_i \in V^k \text{ and } e^k_j \in E^k \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the $N_{\mathrm{nodes}} \times N_{\mathrm{links}}$ *overall application unoriented incidence diagonal block-matrix* A is defined as

$$A = \begin{bmatrix} A^1 & & \\ & \ddots & \\ & & A^k \end{bmatrix}$$

2) Definition of the Decision Variables:

Definition 8. The $N_{\mathrm{CRs}} \times N_{\mathrm{nodes}}$ *decision matrix* $X^{\mathrm{CRs}\to\mathrm{nodes}}$, mapping Application Nodes to CRs, is defined as:

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

Definition 9. The $N_{\mathrm{paths}} \times N_{\mathrm{links}}$ *decision matrix* $X^{\mathrm{paths}\to\mathrm{links}}$, mapping Application Links to NoC Paths, is defined as:

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

Definition 10. The $N_{\mathrm{apps}} \times 1$ *decision vector* r, representing which applications are executed, is defined as:

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

Definition 11. The $N_{\mathrm{nodes}} \times 1$ *decision vector* M, representing which application nodes are reallocated, is defined as:

$$M_i = \begin{cases} 1 & \text{if the Application Node } i \text{ is moved} \\ & \text{from its previously allocated CR} \\ 0 & \text{otherwise} \end{cases}$$

The decision matrices are then vectorized and all vectors are aggregated into one global decision vector $\boldsymbol{x}$:

$$\boldsymbol{x} = \left(\mathrm{vec}(X^{\mathrm{CRs}\to\mathrm{nodes}})^{\mathrm{T}},\ \mathrm{vec}(X^{\mathrm{paths}\to\mathrm{links}})^{\mathrm{T}},\ r^{\mathrm{T}},\ M^{\mathrm{T}}\right)^{\mathrm{T}},$$

where vec is the common vectorization function for matrices: $\mathrm{vec}((a_{i,j})_{1\le i\le m;\ 1\le j\le n}) = [a_{1,1}, \ldots, a_{m,1}, a_{1,2}, \ldots, a_{m,2}, \ldots, a_{1,n}, \ldots, a_{m,n}]^{\mathrm{T}}$.

C. Formulation of the optimization problem

In the formulation provided here, we incorporate the requirement of keeping the execution of the higher priority application over the lower priority ones and executing the lower priority applications whenever there are not enough CRs for the higher priority ones.

1) General form of the optimization problem: As previously mentioned, the allocation problem is formulated as a Pseudo-Boolean problem or a Binary Program of the form:

$$\begin{array}{ll} \text{maximize} & f(\boldsymbol{x}) = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \\ \text{subject to} & M_1\boldsymbol{x} \leq \boldsymbol{b_1} \\ & M_2\boldsymbol{x} = \boldsymbol{b_2} \\ \text{and} & \boldsymbol{x} \text{ is a binary vector.} \end{array}$$

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

2) Objective function: Given the priority of the applications in a descending order i.e. the first application has the highest priority and the N_{apps}-th application has the lowest one, the objective function is used in order to maximize the number of executed applications while minimizing the number of reallocations. The chosen objective function is:

$$\max\left\{ f(\boldsymbol{x}) = \sum_{k=1}^{N_{\text{apps}}} \alpha_k \cdot r_k - \sum_{j=1}^{N_{\text{nodes}}} M_j \right\}, \tag{1}$$

where

$$\begin{aligned} \alpha_{N_{\text{apps}}} &= N_{\text{nodes}}^{N_{\text{apps}}} + N_{\text{nodes}} + 1 \\ \text{and } \forall k < N_{\text{apps}} &: \\ \alpha_k &= \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + \sum_{l=k}^{N_{\text{apps}}} N_{\text{nodes}}^{l} + N_{\text{nodes}} + 1. \end{aligned} \tag{2}$$

The first part of the coefficients α_k are chosen so that a high-priority application will always be executed before all the lower-priority ones. The second term of the coefficient α_k ensures that if moving an application requires the other applications to be moved as well, all required applications will always be moved rather than dropping this application. The last two terms of the coefficient α_k ensure that an application will always be moved rather than dropped. These statement are proven in the Appendix.

3) Constraints: Although some of the constraints are inherited from [5], we state them here for completeness.

a) Binary variables: All the decision variables are binary i.e. their value must be either 0 or 1.

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.

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

$$\forall i = 1, \ldots, N_{\text{CRs}}, \ \sum_{j=1}^{N_{\text{nodes}}} X_{ij}^{\text{CRs}\to\text{nodes}} \leq 1. \tag{3}$$

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

$$\forall i = 1, \ldots, N_{\text{nodes}}, \ \sum_{j=1}^{N_{\text{CRs}}} X_{ji}^{\text{CRs}\to\text{nodes}} = r_{N(i)}. \tag{4}$$

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

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

$$\forall i = 1, \ldots, N_{\text{paths}}, \ \sum_{j=1}^{N_{\text{links}}} X_{ij}^{\text{paths}\to\text{links}} \leq 1. \tag{5}$$

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

$$\forall i = 1, \ldots, N_{\text{links}}, \ \sum_{j=1}^{N_{\text{paths}}} X_{ji}^{\text{paths}\to\text{links}} = r_{L(i)}. \tag{6}$$

  $L(i)$ 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.

$$X^{\text{CRs}\to\text{nodes}}\ A = G\ X^{\text{paths}\to\text{links}}. \tag{7}$$

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

$$\begin{aligned} \forall i \in [\![1, N_{\text{CRs}}]\!],\ \forall j \in [\![1, N_{\text{nodes}}]\!],\ \text{s. t. } X_{\text{old}\ ij}^{\text{CRs}\to\text{nodes}} = 1, \\ \left(1 - r_{N(j)}\right) + M_j + X_{ij}^{\text{CRs}\to\text{nodes}} = X_{\text{old}\ ij}^{\text{CRs}\to\text{nodes}}, \end{aligned} \tag{8}$$

with $X_{\text{old}}^{\text{CRs}\to\text{apps}}$ be the parameter containing the mapping between CRs and Application Nodes computed during the previous allocation.

This constraint is ignored for the initial allocation.

[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.

e) Faults: A constraint inherited from [7] make the algorithm take into account faulty CRs. Within a CR i:

- If the CR is healthy, any Application Node can be mapped on the CR.
- If the CR is faulty, then no Application Nodes can be mapped on the CR i:

$$\sum_{k=1}^{N_{\text{nodes}}} X_{ik}^{\text{CR}\rightarrow\text{apps}} = 0. \tag{9}$$

The detection of this fault is either assumed for the model or detected by the voter using the majority rule described in V-B.

f) Spatial Orientation: To ensure correct orientation of applications, a set of constraints used in [7] is also needed. In order to enforce this, the numbering of the CRs on the NoC is used. For example, as illustrated in Fig. 3, a CR has always a number difference of -1 with its right neighbor and $+N_{/\text{row}}$ with its top neighbor, where $N_{/\text{row}}$ is the number of Tile per row of the NoC ($N_{/\text{row}} = 4$ in our example).

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

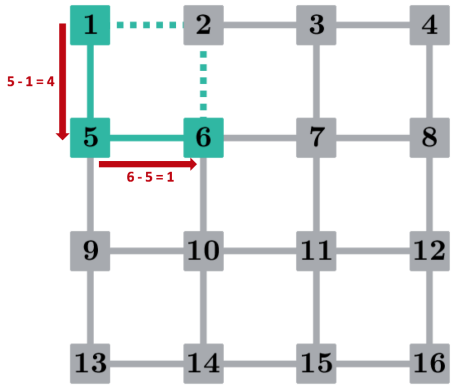
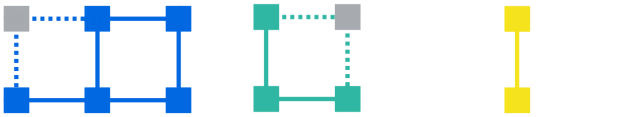


Figure 3: By equating the difference between two CRs' indices allocated to an application to a specific number, the spatial orientation of the application can be enforced.

IV. Algorithm Comparison

The previous modification enables us to compare between the algorithm implemented in [5] and the one proposed here using the model with the three applications shown in Fig. 4.



(a) Application 1, highest priority, critical

(b) Application 2, intermediate priority, non-critical

(c) Application 3, lowest priority, non-critical

Figure 4: The spatial configurations of the three different applications that are to be executed on the platform.

After having forced the same initial configuration for the two algorithms, their behavior are compared for the same fault sequence in Fig. 5.

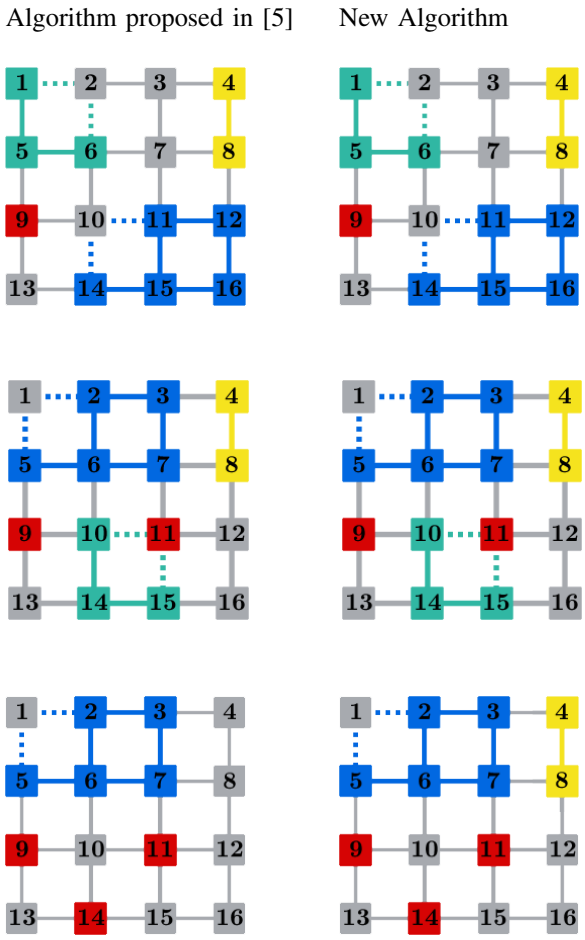


Figure 5: Comparison of the two algorithms for the same CR faults sequence, represented by red squares. When the fault affects the intermediate priority application (green), the first algorithm drops all the lower ones (yellow), unlike the second one, which manages to keep the lower priority applications.

The proposed algorithm uses the remaining CRs of the REDEFINE Fabric better, and more applications can continue to be used, even after a high-priority application is dropped.

V. Experimental Setup

A. Hardware components

To illustrate and demonstrate the capabilities of the new formulation of the allocation algorithm in operational conditions, we choose to implement it on a macroscopic, simplified representation of the REDEFINE architecture, in order to control and maintain operation of a physical system despite the presence of faults.

1) Replication of REDEFINE: Since the REDEFINE architecture is still in development, we chose to build a hardware model of it. In this model, each Tile of the

REDEFINE architecture is represented by a Raspberry Pi computer, as described in our previous work [5]. A REDEFINE Execution Fabric of 4×4 cores is thus replicated with a network of 16 Raspberry Pi computers. An extra Raspberry Pi computer is used to represent the Resource Manager where the allocation algorithm is executed. All of them are connected to a common routing switch in a local area network (LAN). At this stage of the work, we did not try to accurately reproduce the communication protocol used on the REDEFINE Fabric [6].

2) Faults: Two types of faults are considered in this experiment. The first type is computational fault, which randomly affects the computations performed by the Raspberry Pi. We detect this kind of fault by using redundant copies of the considered application combined with a voting system that is described below in Section V-B. The second type of fault is assumed to stop the operation of the Tile it affects. We also assume that this fault can be detected by the Resource Manager. In practice, each time one of these faults affects a Raspberry Pi, the status signal sent by this Raspberry Pi to the Resource Manager is changed to a signal identifying it as faulty. Each of these two kinds of fault can be manually triggered or recovered thanks to a breadboard as seen in Fig. 6 connected to each Raspberry Pi.

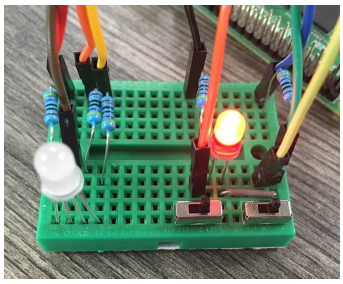


Figure 6: Hardware associated with each Raspberry Pi Tile.
The RGB-LED (bottom-left corner) represents the LED application. The red LED (right side) indicates an healthy Tile when turned on. Each switch is used to trigger one type of fault.

3) Controlled system: The physical system we chose to control with this model of the REDEFINE architecture is a propulsion system, made of an electric fan mounted on a thrust stand (Fig. 7). The fan is commanded by using Pulse width modulation (PWM). The measure of the thrust is used by a simple proportional controller executed on the Fabric to compute the value of the PWM command required to maintain the thrust at a constant value.

An extra Raspberry Pi is used as the micro-controller of the fan: it converts the value measured by the load cell, sends it to the "Execution Fabric" where the appropriate control value is computed, and generates the corresponding PWM signal controlling the fan. It must therefore be noted that although the same hardware representation is used, this Raspberry Pi does not correspond to the same components as the ones used for the Tiles of the Execution Fabric.

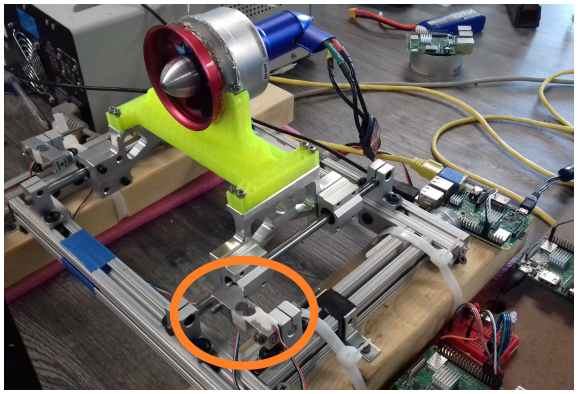


Figure 7: Electric fan mounted on the thrust stand.
The delivered thrust is measured thanks to a load cell on the stand, indicated by the orange circle.

B. Software components

Even if a controller is reallocated to healthy Tiles when it is affected by a fault, because of the time required to compute the new allocation and to actually reallocate the set of tasks, the operation of the fan may be temporarily altered during the reallocation process.

To avoid interruptions in the operation of the fan during reallocations, we use a standard Triple Modular Redundancy (TMR) architecture [12]. Three copies of the controller are executed on the Fabric. Each one separately computes the duty-cycle value of the PWM signal that should be sent to the fan, given the thrust value that they all receive from the sensor. The three values are sent to the Raspberry Pi representing the micro-controller of the fan, where a voting system decides which control output should be used. The vote outputs the result that has been computed by the majority of the controllers, in this case 2 out of 3. Signals are here considered equal if their difference is smaller than a given tolerance. In the case of a fault affecting the output of one of the controllers, the two remaining healthy controllers ensure that the correct value is sent to the fan. The voting system also identifies which controller is not coherent with the two others and informs the Resource Manager of the fault. The reallocation process that we implemented can then take place while providing continuity of service with the two healthy controllers.

To complicate the reallocation tasks, each copy of the controller has been arbitrarily attributed to 2 Application Nodes. Concretely, only one of them is responsible for actual computations. In addition to the 3 redundant controllers, two dummy applications are considered in this experiment: they occupy some Tiles of the Fabric, but do not perform actual computation except changing the voltage in the RGB LED to display their corresponding color. Both have lower priority than the controllers, which represent the safety-critical application in this case. The initial allocation of all these applications on the model is given in Fig. 8, as well as their

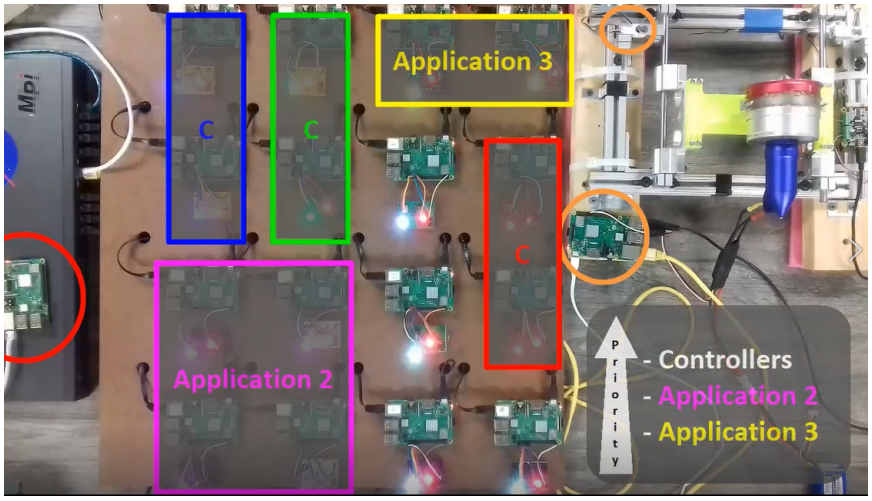


Figure 8: Initial allocation of the applications on the model, with their respective priority.
The Resource manager is identified by the red circle; the small orange circle identifies the load sensor while the big one identifies the extra Raspberry Pi for interactions with the stand.

relative priority. As for the allocation algorithm itself, it is still executed on the Resource Manager, outside of the Fabric.

VI. Results

The evolution of the allocated applications on the experimental system during a fault sequence is shown in Fig. 9, starting from the initial allocation presented in Fig. 8. A link to a recording of the whole experiment is also provided. The experiment shows in particular the ability to recover low priority applications when no more space is available for high priority ones.

The other major characteristic of this experiment is the continuity of service of the fan ensured by the triple redundancy system, even when one of the controller sends a faulty control signal. The graph of the PWM value computed by each controller and the measured thrust of the fan is given in Fig. 10.

In the first part of the graph, we see that controllers adapt the control signal to the measure, when the fan is subjected to perturbations: the PWM signal sent to the fan and the measured thrust evolve in opposite directions. Peaks in the graph of the individual PWM values computed by the controllers correspond to computational errors. Yet, the operation of the fan is maintained in spite of these faults and the delivered thrust remains unaffected.

After approximately 8.5 seconds, the blue controller is dropped, due to a lack of available healthy Tiles. During a few seconds, only 2 controllers are sufficient to operate the fan, until the green controller fails. Indeed, when the only two remaining controllers compute different values, the voting system cannot decide which one is correct, and stops the fan. Nevertheless, when the green controller is recovered, there are enough controllers for the voting system. As a result the fan re-operates again until all controllers are terminated.

VII. Conclusion

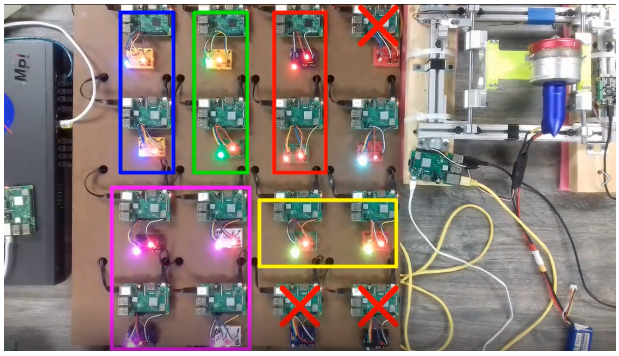
This paper builds upon prior work published at the Digital Avionics Systems Conference and introduces the three improvements below.

A modified formulation of the task allocation algorithm now allows the execution of low priority applications when resources for higher-priority applications are not available anyway. We also added the feature to recover faulty cores and thus use them to run applications that were previously dropped.

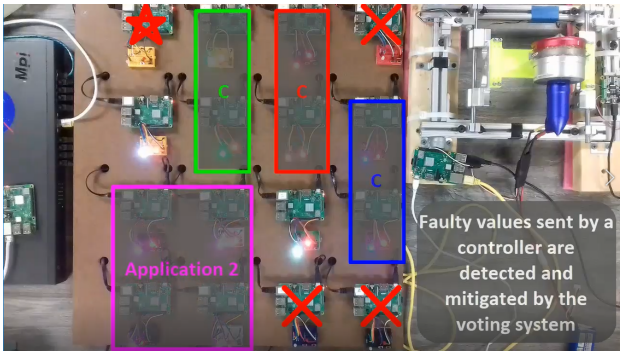
The experiment has also been improved by using the model of REDEFINE to control a propulsion system made of a en electric fan mounted on a stand measuring the delivered thrust. This allowed us to include non-trivial tasks in the set of applications to be reallocated on the model of REDEFINE, that is to say a proportional controller to regulate the thrust of the fan.

The last feature of this work is the adaptation of the traditional triple redundant architecture to a multi-core architecture by using dynamic reallocation in order to maintain the triple redundancy despite core failures. In particular here, it ensures the continuity of service of the fan during the reallocation of one of the three controllers, as shown in our experimental results.

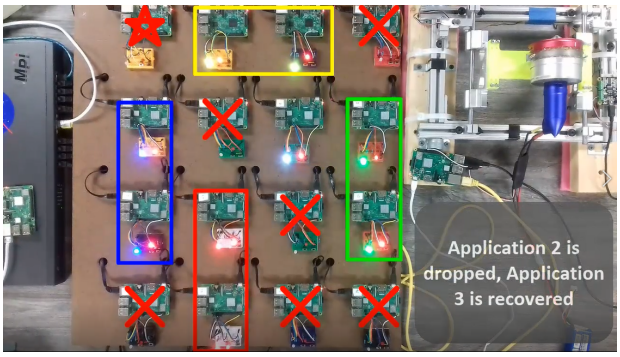
The triple redundancy system would in addition be a conceivable approach to implement a more general fault detection system for other applications than the controller, in particular a task allocator that would be executed on the REDEFINE Fabric itself instead of on an external Resource Manager that can be prone to fail. This approach constitutes a serious option in an effort towards a decentralized task allocation system, where no central element is responsible of the allocation of the tasks on the whole system.



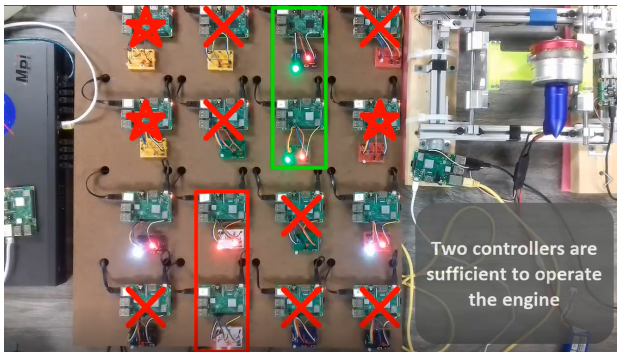
(a) After 3 faults, the algorithm managed to maintain all applications.



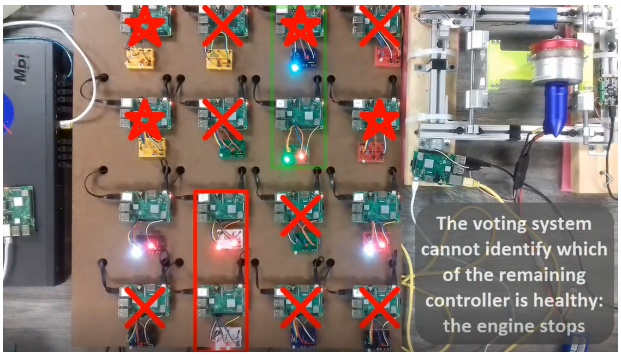
(b) The computational fault affecting a controller is detected and mitigated by the voting system. The lowest-priority application is dropped to reallocate the faulty controller.



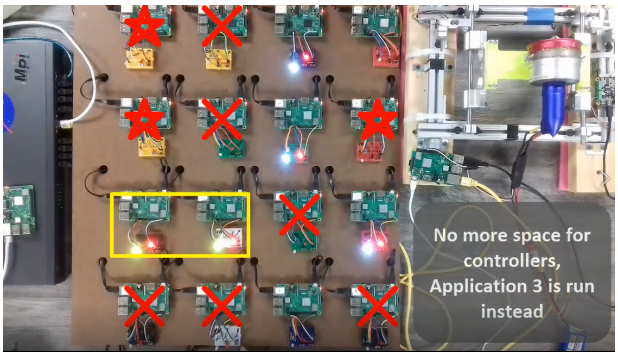
(c) Application 2 is dropped, new free cores are used to run Application 3.



(d) After 10 faults and only two remaining controller, the fan is still operating.



(e) The healthy controller and the affected one send different signals. Since there is no 3rd controller anymore, the vote cannot determine which signal is correct.



(f) A low priority application can run even if safety-critical ones are not operating anymore.

Figure 9: Experimental results.
A red star represents a computational fault and a red cross represents a Tile fault (see paragraph V-A2).
Only major steps are shown here, a full video of the experiment is available on the clickable link:
https://www.dropbox.com/s/m5sekkrlut8mmuj/demo_video_04_19.mp4?dl=0

APPENDIX

This section provides a proof of the statement in III-C. Prior to the Theorem, some useful statements are given as follows

Definition 12. The optimal solution $\boldsymbol{x}^*$ to an optimization problem described in Section III-C2 is a vectorization of the optimal value of all decision variables described in Section III-B2.

Definition 13. For each application $i \in [\![1, N_{\text{apps}}]\!]$, we define the set $\mathcal{N}^i$ as the set of the Application Nodes of application i. Furthermore, $\mathcal{N}^{\mathcal{I}}$ is a set of Application Nodes required to run a set, $\mathcal{I} = \{i_1 \dots i_I\}$, of I applications.

Fact 1. $\forall j \leq N_{\text{apps}}$, $N_{\text{nodes}} \geq \sum_{i=j}^{N_{\text{apps}}} N^i_{\text{nodes}} \geq N^j_{\text{nodes}} > 0$.

Lemma 1. *Let α_k be given by the Eq. 2, then:*

$$\forall i < N_{\text{apps}},\ \forall j \leq N_{\text{apps}},\ i < j\ ;\ \alpha_i > \sum_{k=j}^{N_{\text{apps}}} \alpha_k > 0.$$

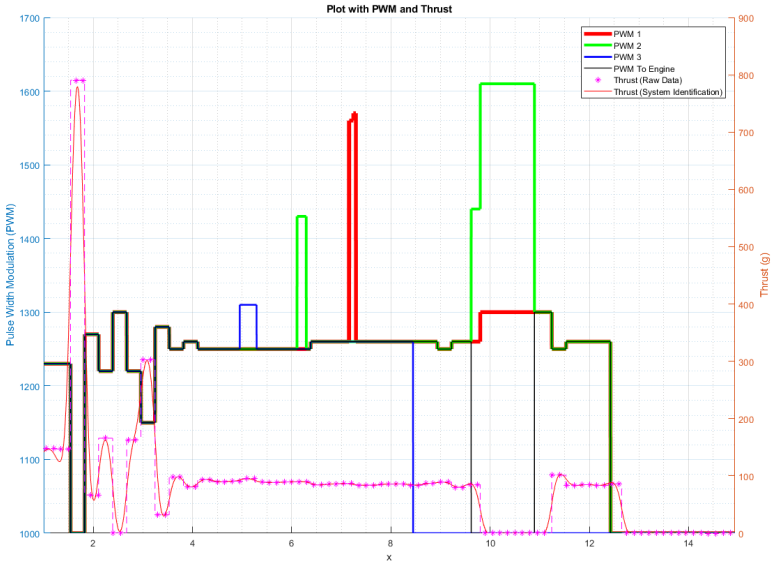


Figure 10: PWM value computed by each controller and measured thrust during operation of the fan.

Proof.

$$\alpha_i = \sum_{l=i+1}^{N_{\text{apps}}} \alpha_l + \sum_{l=i}^{N_{\text{apps}}} N^l_{\text{nodes}} + N_{\text{nodes}} + 1 > 0$$

$$\alpha_i > \sum_{l=i+1}^{N_{\text{apps}}} \alpha_l$$

$$\geq \sum_{k=j}^{N_{\text{apps}}} \alpha_k$$

since $i < j \Rightarrow i + 1 \leq j$. Q.E.D. □

Now, we are ready to state the Theorem.

Theorem 2. *Given the optimization problem described in Section III-C2, and if reallocating the non-lowest priority application i requires other $N_{i'}$ applications to be reallocated as well as another $N_{i''}$ lower priority applications to be dropped, then its optimal solution $\boldsymbol{x}^*$ always reallocates $N_{i'}$ applications and drops $N_{i''}$ lower priority applications in order to reallocate application i if there are not enough CRs to execute that application unless some lower priority ones are dropped.*

Proof. By contradiction, suppose there exists an optimal solution $\hat{\boldsymbol{x}}^*$ that drops the non-lowest priority application i in order to keep other $N_{i'}$ applications and $N_{i''}$ lower priority applications, which means $f(\hat{\boldsymbol{x}}^*) > f(\boldsymbol{x}^*)$. The objective function is

$$f(\boldsymbol{x}) = \sum_{k=1}^{N_{\text{apps}}} \alpha_k \cdot r_k - \sum_{j=1}^{N_{\text{nodes}}} M_j \tag{10}$$

where

$$\alpha_k = \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + \sum_{l=k}^{N_{\text{apps}}} N^l_{\text{nodes}} + N_{\text{nodes}} + 1. \tag{11}$$

Let $\mathcal{I}' = \{i'_1 \dots i'_{N_{i'}}\}$ be the set of the $N_{i'}$ applications required to be allocated, $\mathcal{I}'' = \{i''_1 \dots i''_{N_{i''}}\}$ be the set of the other $N_{i''}$ applications required to be dropped and $\mathcal{I} = \{i\} \cup \mathcal{I}' \cup \mathcal{I}''$. Dropping application i means that the value of r_k corresponding to application i must be zero as well as every element of the decision vector M since no Application Nodes are required to move. Therefore, the objective function is

$$f(\hat{\boldsymbol{x}}^*) = \sum_{k=1|k\notin\mathcal{I}}^{N_{\text{apps}}} \alpha_k \cdot r_k + \sum_{k\in\mathcal{I}'\cup\mathcal{I}''} \alpha_k \cdot 1. \tag{12}$$

On the other hand, for the objective function $f(\boldsymbol{x}^*)$, the value of r_k and M_j(s) corresponding to application i must be one, since it has been reallocated in order to keep it running as well as the ones corresponding the other i' applications. However, the value of r_k and M_j corresponding the other $N_{i'}$ applications must be zero since they have been dropped. Therefore, the objective function is

$$\begin{aligned} f(\boldsymbol{x}^*) = &\sum_{k=1|k\notin\mathcal{I}}^{N_{\text{apps}}} \alpha_k \cdot r_k + \sum_{k\in\mathcal{I}'} \alpha_k \cdot 1 + \alpha_i \cdot 1 \\ &- \sum_{l\in\mathcal{I}'} \left(\sum_{j=1}^{N^l_{\text{nodes}}} 1 \right) - \sum_{j=1}^{N^i_{\text{nodes}}} 1. \end{aligned} \tag{13}$$

Then,

$$
\begin{aligned}
f(\hat{\boldsymbol{x}}^*) - f(\boldsymbol{x}^*) &= \sum_{k\in\mathcal{I}''} \alpha_k - \left(\alpha_i - \sum_{l\in\mathcal{I}'} N^l_{\text{nodes}} - N^i_{\text{nodes}}\right) \\
&= \sum_{k\in\mathcal{I}''} \alpha_k + \sum_{l\in\mathcal{I}'} N^l_{\text{nodes}} + N^i_{\text{nodes}} \\
&\quad - \left(\sum_{l=i+1}^{N_{\text{apps}}} \alpha_l + \sum_{l=i}^{N_{\text{apps}}} N^l_{\text{nodes}} + N_{\text{nodes}} + 1\right) \\
&= \left(\sum_{k\in\mathcal{I}''} \alpha_k - \sum_{l=i+1}^{N_{\text{apps}}} \alpha_l\right) - 1 \\
&\quad + \left(\sum_{l\in\mathcal{I}'} N^l_{\text{nodes}} - N_{\text{nodes}}\right) \\
&\quad + \left(N^i_{\text{nodes}} - \sum_{l=i}^{N_{\text{apps}}} N^l_{\text{nodes}}\right).
\end{aligned} \tag{14}
$$

By Fact 1 and Lemma 1,

$$
\begin{aligned}
f(\hat{\boldsymbol{x}}^*) - f(\boldsymbol{x}^*) &= -\sum_{k>i|k\notin\mathcal{I}''} \alpha_k - 1 \\
&\quad - \sum_{l\notin\mathcal{I}'} N^l_{\text{nodes}} - \sum_{l=i+1}^{N_{\text{apps}}} N^l_{\text{nodes}} \qquad (15) \\
f(\hat{\boldsymbol{x}}^*) - f(\boldsymbol{x}^*) &< 0 \\
f(\hat{\boldsymbol{x}}^*) &< f(\boldsymbol{x}^*)
\end{aligned}
$$

which is a contradiction. Thus, the statement in Section III-C is proven. Q.E.D.

□

REFERENCES

[1] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, Oct 2007, pp. 2.A.1–1–2.A.1–10.

[2] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs—combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, Oct 2012.

[3] N. Neves, N. Sebastião, D. Matos, P. Tomás, P. Flores, and N. Roma, "Multicore simd asip for next-generation sequencing and alignment biochip platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1287–1300, July 2015.

[4] Y. Lu, H. Zhou, L. Shang, and X. Zeng, "Multicore parallelization of min-cost flow for cad applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1546–1557, Oct 2010.

[5] L. Sutter, T. Khamvilai, P. Monmousseau, J. B. Mains, E. Feron, P. Baufreton, F. Neumann, M. Krishna, S. K. Nandy, R. Narayan, and C. Haldar, "Experimental allocation of safety-critical applications on reconfigurable multi-core architecture," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, Sep. 2018, pp. 1–10.

[6] 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.

[7] 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)*, Sep. 2017, pp. 1–10.

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

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

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

[11] "GLPK reference manual," *GNU Linear Programming Kit*, 2012, https://www.gnu.org/software/glpk/TOCdocumentation.

[12] C. M. K. Israel Koren, *Fault tolerant systems*. Morgan Kaufmann Publishers, 2007.